An Empirical Analysis of CryptGenRandom in Windows XP SP3 and Its Historical Relevance to Bitcoin 0.1.5

Melik Lemariey
Consultant
France
f561bc6a712a@pm.me
https://meliklemariey.com

Abstract

We present an empirical reconstruction of the random-generation path linking `CryptGenRandom` on Windows XP SP3 to a cryptographic environment consistent with Bitcoin 0.1.5 on Windows. The study relates this Windows RNG path to Bitcoin 0.1.5 through OpenSSL 0.9.8h: Bitcoin consumes random bytes through `RAND_bytes`, while OpenSSL's Windows RAND state may be seeded through `RAND_poll`/`RAND_screen`, including `CryptGenRandom` as one system source. The Windows-side path then proceeds through the provider `rsaenh` and kernel `KSecDD`.

The methodology combines dynamic instrumentation under WinDbg, paired memory captures, controlled injections, and offline replay in Python. On the kernel side, we validate the construction of a `0x258`-byte pool, its segmentation into four `0x96`-byte blocks, and its consumption by `VeryLargeHashUpdate`. For captured pools, the replay reproduces `seedbase_after` bit-for-bit.

On the provider side, we identify effective uses of MD4, SHA-1, RC4, a circular buffer, and the IOCTL `0x390008`. The final stage of the pipeline is experimentally closed: `state20` and `aux20` feed a final block replayed offline through SHA-1 compression and 160-bit additions, producing exactly `out40`. The first `0x20` bytes are then copied into the user buffer returned by `CryptGenRandom`.

The OpenSSL and Bitcoin layers are treated separately from the Windows RNG reconstruction. The `randwin` artifacts structurally validate the Windows-side records collected by OpenSSL's `RAND_poll` layer, while the `ssleay` replay validates the post-stir `ssleay_rand_bytes` generation loop and reproduces the observed 32-byte output. A wallet-level check verifies that this output is consistent with the corresponding WIF encoding and Bitcoin P2PKH address; an additional check with the historical Bitcoin 0.1.5 client confirms that the generated `wallet.dat` is recognized with the same uncompressed address.

This work is presented as a pre-publication based on an empirical approach prioritizing trace-grounded and reproducible results. Reproducibility artifacts are provided in a public repository containing executable validation scripts and captured inputs for the closed components of the pipeline. The main remaining open component is the upstream provider-state mapping from `seedbase_after` and persistent provider state to `state20`. The study does not claim a practical attack against Bitcoin or a complete entropy assessment; it documents a historically observable and partially bit-accurate replayable path.

Author's note. This work was conducted independently by Melik Lemariey, acting as an independent consultant. All experiments, analyses, implementations, and possible errors are solely the responsibility of the author. Discussions with external researchers contributed to refining certain ideas, but do not constitute validation, endorsement, peer review, or co-authorship. The code and experimental tooling used in this study were developed by the author, unless explicitly stated otherwise. The author is not affiliated with any academic, industrial, or governmental institution in relation to this work.

Large language models were used as an auxiliary tool to accelerate drafting, language refinement, editorial organization, and non-authoritative technical discussion. They were also used to help structure reasoning, identify possible coherence issues, and compare alternative formulations. They were not used to automate the experimental workflow or to generate validation results. All experimental procedures, data collection, reverse-engineering analysis, instrumentation decisions, replay implementations, and validation results were designed, performed, checked, and interpreted manually by the author.

Keywords. CryptGenRandom; Windows XP; CryptoAPI; `rsaenh`; `KSecDD`; random number generation; reverse engineering; dynamic instrumentation; offline replay; OpenSSL; `RAND_poll`; `RAND_bytes`; Bitcoin.

1 Introduction

1.1 Context

`CryptGenRandom` long served as the standard interface exposed to Windows applications for obtaining pseudo-random bytes for cryptographic use. On Windows XP, this interface lies at the junction of several layers: the calling application, CryptoAPI, the cryptographic provider `rsaenh`, and, under certain conditions, kernel routines associated with `KSecDD` that are responsible for collecting and mixing system information.

This historical architecture is important for two reasons. First, it predates the widespread adoption of modern standardized DRBG constructions. Second, it was used by real-world software from the 2008–2010 period through libraries such as OpenSSL. Understanding its effective behavior therefore makes it possible to document an actually deployed cryptographic stack, rather than only a construction described abstractly.

1.2 Problem Statement

The central problem of this study is to reconstruct, as far as possible through direct observation and offline replay, the effective chain linking the observed system sources to the output returned by `CryptGenRandom` on Windows XP SP3. The study then examines how this output is integrated into the `RAND_poll`/`RAND_bytes` path of OpenSSL 0.9.8h and into the experimental context of Bitcoin 0.1.5 on Windows.

The objective is neither to provide a complete formal security proof nor to demonstrate a practical compromise of historical keys. Rather, it is to establish an experimental mapping of the real pipeline, from system calls and the `rsaenh` provider to the seeding of OpenSSL's pseudo-random state and its observable use in an instrumented historical Bitcoin client.

This study strictly distinguishes:

— behaviors directly observed during execution;

— transformations replayed bit-for-bit offline;

— experimental relationships between `CryptGenRandom`, `RAND_poll`, `RAND_bytes`, and Bitcoin 0.1.5;

— structural models strongly constrained by traces;

— points that remain open.

1.3 Approach

Our approach combines dynamic instrumentation under WinDbg, targeted disassembly, controlled injection campaigns, and offline validation in Python. Each important stage of the pipeline is treated as an experimental object: inputs are captured, outputs are observed, and the computation is then reconstructed in an external environment whenever possible.

This methodology avoids two pitfalls. The first would be to limit the analysis to a static reconstruction of the binary, without guarantees on the paths actually executed. The second would be to rely only on dynamic traces without a computable model. The chosen approach connects the two: traces determine the model, and replay then tests its precision.

1.4 Scope

The study focuses on Windows XP SP3, the `rsaenh` provider, and the paths observed in the experimental environment described below. The results should not be automatically generalized to other Windows versions or to other providers. The addresses given in the text correspond to the binaries and load addresses observed during the campaigns. They serve as experimental anchors rather than stable interfaces.

1.5 Summary of Results

The main results can be summarized as follows. The kernel phase is followed from the sequence of calls to `ZwQuerySystemInformation` to the construction of an aggregated `0x258`-byte pool. The campaigns make it possible to locate the relevant QSI contributions within this pool and to distinguish regions corresponding to zeros, metadata, or non-essential residual data. For captured pools, the `Pool` → `VeryLargeHashUpdate` → `seedbase_after` transformation is reconstructed and replayed offline with bit-for-bit agreement for the observed instances, providing an experimental validation of this stage of the pipeline.

On the provider side, several primitives and internal mechanisms are observed, including MD4, RC4-related operations including a KSA phase, and SHA-1 compressions on 64-byte blocks. The local input to the final block is stabilized before `68027101`. The final block $H(\texttt{state20}, \texttt{aux20})$ is closed bit-for-bit and produces `out40`. The first `32` bytes of `out40` are then copied into the user buffer returned by `CryptGenRandom`.

The last substantial unresolved point therefore concerns neither `VeryLargeHashUpdate`, nor the final block, nor the user-buffer copy. It concerns the upstream construction of the persistent provider structure that feeds `state20`, in particular its initialization and update before the final projection into the observed global slot.

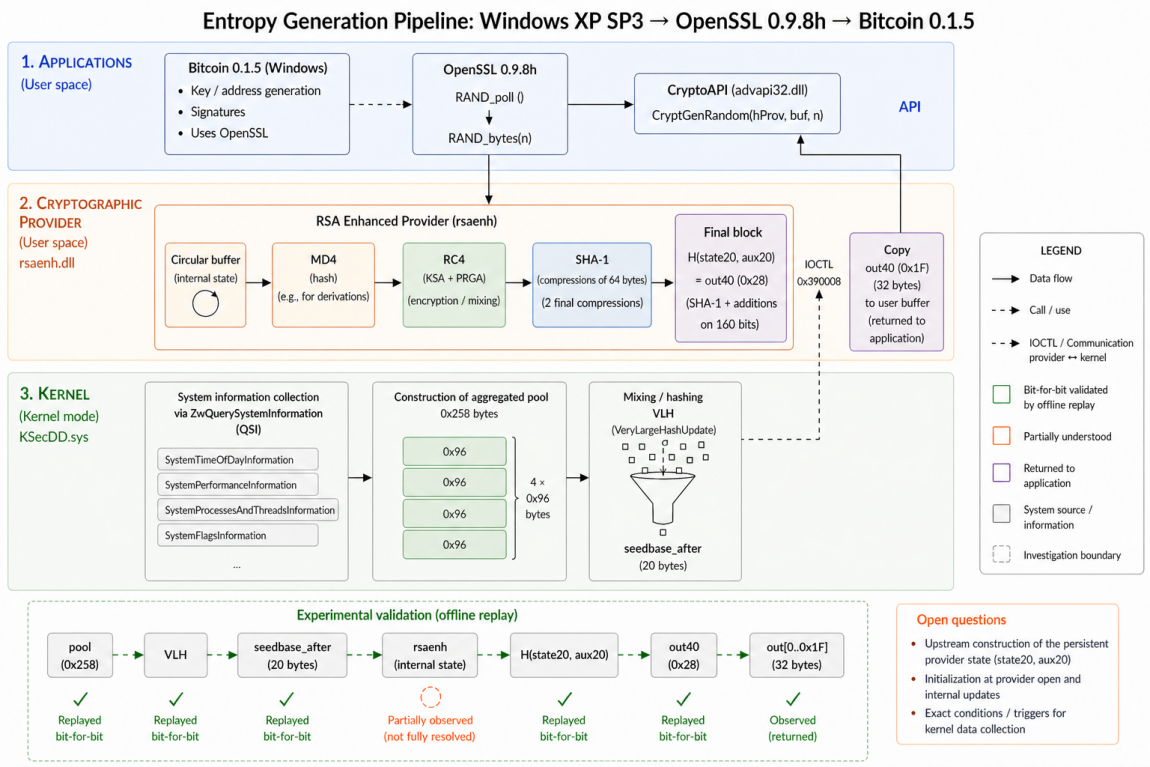

Figure 1 – End-to-end view of the `CryptGenRandom` pipeline on Windows XP. The kernel collects system information through `ZwQuerySystemInformation` and aggregates it into a `0x258`-byte pool, which is processed by `VeryLargeHashUpdate` to produce `seedbase_after`. On the provider side, the upstream construction of the persistent provider state feeding `state20` remains partially open. The closed final stage starts from the observed inputs `state20` and `aux20`, which are combined through a FIPS 186–style construction using two experimentally validated SHA-1 compression rounds to produce `out40`. The final `CryptGenRandom` output corresponds to the first `32` bytes of this value.

2 Related Work and Positioning

2.1 Work on the Windows RNG

The work of Dorrendorf, Gutterman, and Pinkas [3] constitutes a major reference regarding the Windows pseudo-random generator. Their study highlights an architecture notably relying on SHA-1 and RC4, as well as mechanisms for accumulation, mixing, and state update. It provides an essential framework for understanding the general structure of the system and the properties of its internal state.

Our study follows this line of work, but with a different perspective. Rather than proposing only a structural reconstruction of the generator, we aim to connect the actually executed paths to concrete memory captures, and then validate certain stages through offline replay. The expected result is therefore not a more abstract description of the generator, but an experimental decomposition of the observed transformations, with bit-level comparison between computed and captured values.

This methodological difference is important. A static or semi-static description makes it possible to identify primitives and overall structure; an empirical reconstruction makes it possible to determine which paths are actually executed, which values propagate across layers, and which transformations can be reproduced outside the instrumented system.

2.2 Applied Cryptography and Real-World Implementations

This study also fits within a tradition of applied cryptography attentive to implementation details. Ferguson and Schneier, in *Practical Cryptography*, emphasize the importance of randomness quality, system integration, and concrete implementation choices in the effective security of cryptographic systems [5]. This perspective complements more encyclopedic references on cryptographic primitives such as *Applied Cryptography* [11].

This point is central for analyzing `CryptGenRandom` on Windows XP. Identifying SHA-1, MD4, or RC4 in a binary is not sufficient to understand the generator. It is necessary to determine how these primitives are invoked, with which inputs, in which order, across which kernel/provider boundaries, and which values are effectively exposed to the application.

This work therefore adopts an experimental applied cryptography perspective: primitives are important, but their actual composition, initialization, state evolution, and effective use are equally critical.

2.3 Entropy Failures in Real Systems

Several works have shown that weaknesses in random number generators can have observable effects in deployed cryptographic artifacts. The study by Heninger et al. [7] illustrates this approach at scale: weak public keys can reveal entropy issues in real systems. Other work on Linux generators [6] has similarly emphasized the importance of internal state, entropy collection, and conditioning.

These approaches are complementary to ours. They highlight externally observable effects of flawed randomness, whereas this study aims to reconstruct the internal pipeline of a specific historical generator. The objective here is not to infer defects from public outputs, but to document the internal transformations linking observed system sources, intermediate states, and the output returned to the application.

2.4 Modern DRBGs and NIST Standards

The NIST SP 800-90A/B/C recommendations [8, 9, 10] formalize modern random generation constructions by separating entropy sources, conditioning, state update, and deterministic

generation. These models provide a rigorous vocabulary for describing the components of a cryptographic generator.

However, they do not directly describe historical implementations deployed in earlier operating systems. Windows XP belongs to a transitional period: the system combines known primitives, persistent states, circular buffers, kernel calls, and transformations executed within the provider. An empirical analysis is therefore necessary to avoid projecting a modern DRBG model onto an implementation that does not strictly follow it.

In this study, the DRBG vocabulary is used to describe functional roles—collection, conditioning, state, generation—but is not assumed to correspond to a strict structural equivalence. The observed blocks in Windows XP are described based on traces rather than a presumed normative model.

2.5 Dual EC, RC4/TLS, and the Importance of Concrete Implementations

The controversies surrounding `Dual_EC_DRBG` originated in 2007, when Shumow and Ferguson showed that a particular choice of parameters could introduce a trapdoor enabling prediction of the generator [12]. This observation highlighted that a generator compliant with a standard specification may still exhibit structural weaknesses. Subsequent work demonstrated the practical exploitability of this mechanism in real-world contexts, notably in TLS [2].

A related lesson emerges from the analysis of RC4 in TLS. While RC4 was long considered suitable for practical use, its statistical biases were eventually shown to be exploitable in protocol contexts, enabling partial recovery of secret material under realistic conditions [1]. These results do not rely on structural backdoors, but on subtle deviations from ideal randomness that become significant when amplified by protocol usage.

Taken together, these examples emphasize that the security of randomness sources cannot be assessed solely at the level of abstract specifications. Parameters, implementation details, and integration contexts all play a critical role in determining effective security properties.

In the case studied here, the goal is not to demonstrate a backdoor or a practical attack comparable to `Dual_EC_DRBG` or RC4-based TLS attacks. Rather, it is to show that the effective behavior of a historical RNG cannot be reduced to a high-level description. The boundaries between kernel, provider, and application are precisely where dynamic analysis provides additional insight: they determine which data are collected, how they are mixed, when the internal state is updated, and which values are ultimately exposed to the caller.

2.6 Pre-Bitcoin Electronic Cash and Historical Context

The connection to Bitcoin also requires placing this study within the broader history of electronic cash. Prior to Bitcoin, numerous cryptographic works explored digital cash systems, blind signatures, payment anonymity, offline spending, and the double-spending problem.

In this context, Ferguson proposed extensions to *single-term coins*, including coins that can be spent a bounded number of times without revealing the user's identity, while still enabling identification in case of abuse [4]. Such work illustrates the longstanding nature of questions related to digital currency, double spending, and trust in concrete cryptographic mechanisms.

This study does not address electronic cash protocols themselves. It focuses on a different layer: randomness generation in a historical software stack used by applications in the 2008–2010 period. The connection is therefore contextual rather than protocol-level. Pre-Bitcoin work provides historical background, while this analysis documents a system component that may contribute to cryptographic initialization in a historical monetary application.

2.7 OpenSSL, Bitcoin, and Historical Relevance

Bitcoin 0.1.5 relies on OpenSSL 0.9.8h for its cryptographic operations. On Windows, OpenSSL may use `CryptGenRandom` during `RAND_poll` or `RAND_screen` to seed its internal pseudo-random

state. Bitcoin later consumes random bytes from this OpenSSL state through `RAND_bytes`. This establishes a concrete historical relevance for studying `CryptGenRandom` on Windows XP, while preserving the distinction between OpenSSL state seeding and OpenSSL byte generation:

$$\begin{aligned}\texttt{Bitcoin 0.1.5} &\rightarrow \texttt{OpenSSL 0.9.8h}\\ &\rightarrow \texttt{RAND\_bytes}\\ &\leftarrow \texttt{OpenSSL RAND state},\end{aligned}$$

where the OpenSSL RAND state may have previously been seeded through:

$$\begin{aligned}\texttt{RAND\_poll / RAND\_screen} &\rightarrow \texttt{CryptGenRandom}\\ &\rightarrow \texttt{Windows XP RNG}.\end{aligned}$$

This observation does not constitute a claim of attack against Bitcoin nor a proof of key compromise. It simply shows that a historical monetary application, built with OpenSSL on Windows, may follow an execution path in which `CryptGenRandom` contributes to its initial cryptographic state. This justifies the relevance of empirically documenting this generator.

2.8 Positioning

This work lies at the intersection of several approaches: static reconstructions of historical systems, formal analyses of modern generators, and empirical studies of entropy failures observed through external artifacts. It adopts a trace-driven methodology in which each claim is explicitly qualified according to its status: direct observation, bit-level replay, trace-constrained model, or open question.

This framework makes it possible to clearly separate validated segments from unresolved components. In the current state of reconstruction, the kernel chain `ZwQuerySystemInformation` $\rightarrow$ pool $\rightarrow$ `VeryLargeHashUpdate` $\rightarrow$ `seedbase_after` is experimentally validated through offline replay on captured data. Similarly, the terminal part of the provider, from `state20` and `aux20` to `out40`, and then to the user-level copy, is reproduced bit-for-bit.

The remaining uncertainties are concentrated in the upstream stages of the provider, particularly the initialization and update of persistent structures before their projection into `state20`.

The contribution of this work is neither to propose a new abstract model of pseudo-random generation nor to demonstrate a practical exploit. It consists in transforming a complex historical system into an experimentally instrumented pipeline, in which several segments are validated through bit-level replay and the remaining uncertainties are explicitly identified and delimited.

Studied component	Experimental contribution	Status
Execution chain	Instrumentation of application → CryptoAPI → `rsaenh` → `KSecDD` → user output.	Observed
Kernel pool	`0x258`-byte pool captured before `VeryLargeHashUpdate` and replayed through the kernel-side hashing path.	Validated
QSI-to-pool attribution	Stable regions are identified for several `ZwQuerySystemInformation` classes, while some pool regions remain fragmentary or unattributed at byte level.	Partially attributed
`VeryLargeHashUpdate`	Offline replay from captured pool to `seedbase_after`.	Bit-exact
Provider `rsaenh`	MD4, SHA-1, RC4/KSA/PRGA, circular buffer, and IOCTL `0x390008`.	Observed / partial
Local input	`AFTER_LOCAL` = `aux20` before final block.	Closed
Final block H	Replay of $H(\texttt{state20}, \texttt{aux20}) = \texttt{out40}$ using two SHA-1 compressions and 160-bit additions.	Bit-exact
User copy	`CryptGenRandom` returns `out40[0:0x20]` for observed `0x20`-byte calls.	Closed
Lock G	Upstream construction of persistent structures feeding `state20`.	Open / localized
OpenSSL / Bitcoin	Bitcoin consumes OpenSSL `RAND_bytes`; on Windows, the OpenSSL RAND state may be seeded through `RAND_poll`/`RAND_screen`, including `CryptGenRandom` as one system source.	Experimental relevance

Table 1 – Summary of experimental contributions and their validation status.

3 Contributions

This study provides the following experimental contributions, summarized in Table 1. The subsections below detail each point.

We document an execution chain spanning user space, the `rsaenh` provider, and the `KSecDD` kernel. Observation points include `CryptGenRandom`, `ZwDeviceIoControlFile`, `ZwQuerySystemInformation` calls, `VeryLargeHashUpdate`, internal provider primitives, the final generation block, and the copy to the user buffer.

This instrumentation makes it possible to relate events observed at different execution levels: system calls, kernel buffers, provider structures, intermediate states, and the output returned to the application. The addresses referenced in the study are used as experimental anchors for the observed binaries.

3.1 Characterization of the kernel pool

We confirm a stable sequence of `ZwQuerySystemInformation` calls:

$$5,\ 3,\ 7,\ 2,\ 0x21,\ 0x2d,\ 8,\ 0x17.$$

Manual and automated campaigns show that the `0x258`-byte kernel pool contains several distinct regions: data originating from `ZwQuerySystemInformation` classes, a prefix transformed by a helper consistent with `bswap32`, zeroed regions, allocation metadata, and residual uninterpreted data. Classes `3`, `7`, `2`, `0x21`, and `0x2d` are observed as exact copies at stable offsets in validated captures.

The key point is that this characterization is sufficient to reconstruct the buffer effectively consumed by `VeryLargeHashUpdate` in the studied campaigns. Some bytes of the raw buffer may remain described as metadata, padding, or semantically uninterpreted residues; this does

not prevent full validation of the VeryLargeHashUpdate computation as long as the captured pool is available and its consumption is replayed bit-for-bit.

3.2 Full replay of VeryLargeHashUpdate from captured pools

For captured pools, the VeryLargeHashUpdate transformation is reconstructed and replayed offline with bit-exact correspondence. The four intermediate values seed0' to seed3', as well as seedbase_after, match the values observed in validated campaigns.

This contribution experimentally closes the sub-problem:

$$pool_{0x258} \longrightarrow \texttt{VeryLargeHashUpdate} \longrightarrow \texttt{seedbase\_after}.$$

The scope of this statement is precise: it does not imply that every byte of the raw buffer receives a unique semantic interpretation. It means that, for validated runs, the data actually consumed by VeryLargeHashUpdate is known and that the transformation to seedbase_after is fully replayable offline.

3.3 Observation of provider primitives

On the provider side, the study identifies and documents several internal primitives involved in the observed path. Traces reveal calls and transformations consistent with MD4 hashing operations, SHA-1 processing, and RC4-like mechanisms including phases analogous to KSA and PRGA.

These elements are not treated as purely static signatures. They are tied to observed execution points, concrete buffers, and measurable memory effects. This contribution makes it possible to distinguish primitives effectively encountered in campaigns from abstract descriptions of the provider.

3.4 Local validation of MD4 and RC4/KSA routines

Provider campaigns show that MD4 routines are used in the internal path of rsaenh. Observed points cover initialization, update, finalization, and the MD4 core transformation. Experimental results confirm the presence of internal mixing stages within the provider before the final block, without implying that the full upstream derivation of state20 is completely closed.

Similarly, RC4-like routines, including a KSA phase and a generation phase, are observed and replayed in isolation. At the current stage of reconstruction, these elements are validated as internal components of the provider, but their full integration into the upstream construction of state20 remains distinct from the closed final block.

3.5 Identification of a non-finalizing internal use of SHA-1

The final generation block uses the SHA-1 compression function on 64-byte blocks, without corresponding to a complete SHA-1 invocation with padding, length encoding, and finalization as in standard hashing APIs. It is an internal use of the SHA-1 compression function, embedded in a construction combining 160-bit additions and successive compression steps.

This distinction is important: the final block is not modeled as a simple SHA1(message), but as an internal transformation based on sha1_compress applied to blocks constructed by the provider.

3.6 Closure of the provider local input

Traces show that the local buffer used by the final block is already finalized at the AFTER_LOCAL stage. In paired runs, no modification of this buffer is observed between AFTER_LOCAL and the entry to the final block. We denote this relation:

$$\texttt{AFTER\_LOCAL} = \texttt{aux20}.$$

This relation closes the direct projection of the local buffer into the auxiliary input of the final block.

3.7 Closure of the final block H

From `state20` and `aux20` captured at the entry of the final block, the computation is replayed offline and reproduces exactly `out40`. This step closes the sub-problem:

$$H(\texttt{state20}, \texttt{aux20}) = \texttt{out40}.$$

The validated model uses two calls to the SHA-1 compression function on `64`-byte blocks constructed by the provider, together with big-endian 160-bit additions. The `40` bytes produced by the replay match the `40` bytes observed after execution of the provider block.

3.8 Validation of the user copy

For observed calls requesting `0x20` bytes, the output returned to the caller corresponds to the first `32` bytes of `out40`:

$$\texttt{CryptGenRandom}_{out} = \texttt{out40}[0 : 0x20].$$

This step closes the final observable projection between the internal provider buffer and the output returned to the caller.

3.9 Reduction of the lock G

Recent campaigns further localize the remaining open point G. We show that `state20` is copied into a global slot via an explicit `20`-byte memory copy, and that the immediate source can be traced back to a global provider structure. The remaining uncertainty therefore concerns the upstream initialization and update of this persistent structure, not the final generation stage.

The remaining lock must therefore be formulated precisely: it does not concern `VeryLargeHashUpdate`, the immediate projection of `state20` into the final block, the computation H, or the user copy. It concerns the prior construction of certain persistent provider structures.

3.10 OpenSSL and Bitcoin relevance

We recall and experimentally verify that, on Windows, OpenSSL 0.9.8h uses `CryptGenRandom` in `RAND_poll`. The observed chain is therefore relevant for historical software relying on OpenSSL, in particular Bitcoin 0.1.5 on Windows.

This contribution concerns the execution path and the historical integration of the cryptographic stack. It does not constitute a claim of practical key compromise, nor a proof that historical Bitcoin keys could be derived from the observations presented here.

4 Experimental Methodology

4.1 General Principle

The methodology is based on an alternation between dynamic observation and offline reconstruction. Execution traces are used to identify relevant points, capture buffers, and determine the execution chains actually followed. Replay scripts are then used to verify whether the reconstructed transformations reproduce the observed values.

This distinction is central: a routine may be statically identified as SHA-1, MD4, or RC4, but only the capture of inputs, outputs, and memory effects makes it possible to establish that the observed primitive effectively participates in the studied path with the data under consideration.

The methodological objective is therefore not merely to identify cryptographic primitives in the binaries, but to connect concrete data captured during execution to transformations replayable offline.

4.2 Experimental Environment

The primary target system is a Windows XP SP3 virtual machine configured for kernel debugging. The host system is a Linux machine running QEMU/KVM. Debugging is performed from a Windows 10 machine running WinDbg, connected to the target system through a virtual serial channel at 115200 baud.

In addition, campaigns were conducted on a physical hard drive from a 2006–2009-era laptop running Windows XP. These tests aim to compare the behavior of the generator in a native hardware environment.

No structural difference was observed in the construction and aggregation of the kernel pool between the virtualized environment and the physical environment. The observed variations concern only the speed of system-information collection.

Furthermore, some sporadic errors observed in the virtualized environment, notably during the collection of large volumes of system information, were not reproduced on the physical machine.

This configuration makes it possible:

— to reboot the environment under comparable conditions;

— to capture kernel, provider, and user-space buffers;

— to place breakpoints on kernel and provider locations;

— to replay campaigns under documented experimental conditions.

The reported results should be understood within this precise experimental framework. The addresses, offsets, and behaviors described correspond to the binaries and load configurations observed in this environment.

4.3 Triggering

A minimal program triggers repeated calls to `CryptGenRandom` in order to obtain simple and controllable traces. Additional tests are performed with OpenSSL 0.9.8h and Bitcoin 0.1.5 to verify that the observed paths are not specific to the test program.

In the OpenSSL case, the objective is to confirm the use of `CryptGenRandom` in `RAND_poll` on Windows. In the Bitcoin 0.1.5 case, the objective is to connect this behavior to a historical software application using OpenSSL for cryptographic initialization.

4.4 WinDbg Instrumentation

The WinDbg scripts follow conservative rules: MASM expression evaluator, simple breakpoints, captures via `.writemem`, and `go` wrappers ending with `gc`. This discipline reduces errors related to complex inline commands and facilitates campaign reproducibility.

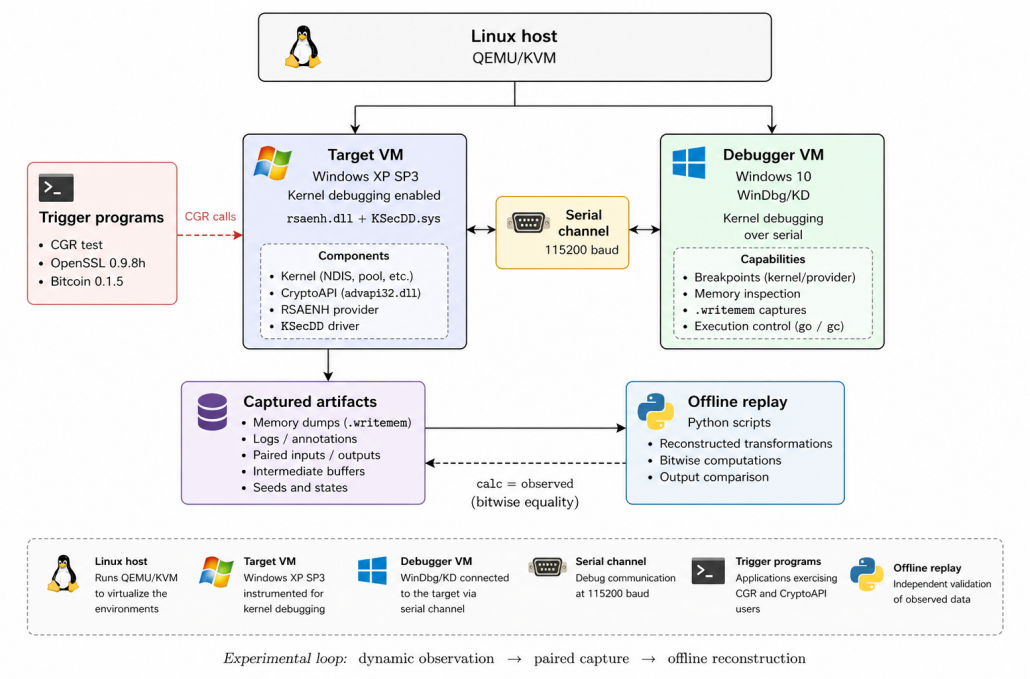

Figure 2 – Experimental laboratory architecture. The Windows XP SP3 target system runs under QEMU/KVM on a Linux host and is configured for kernel debugging. A Windows 10 virtual machine runs WinDbg/KD and communicates with the target through a logical serial channel (UART emulation) at 115200 baud. Test programs (CGR, OpenSSL, Bitcoin) trigger execution, while captured artifacts are replayed offline for bit-level validation.

The main instrumentation points include:

— `ZwQuerySystemInformation` to capture system classes;
— `VeryLargeHashUpdate` to capture the pool, intermediate buffers, and seeds;
— `ZwDeviceIoControlFile` and the IOCTL `0x390008`;
— the provider's MD4, SHA-1, and RC4 routines;
— the KSA-like and PRGA-like stages observed in the provider;
— the final block entry point;
— provider copy sites for final and intermediate copies.

Captures are paired by run when necessary: an input, an output, and intermediate buffers are compared only when they originate from the same execution, or from a campaign explicitly structured to allow such correlation.

4.5 Instrumentation of Application Components

In addition to kernel and provider instrumentation, some campaigns rely on lightweight instrumentation of application components in order to connect low-level observations to high-level calls.

On the OpenSSL 0.9.8h side, targeted instrumentation was added to `md_rand.c` and `rand_win.c` to observe calls to `RAND_poll` and `RAND_bytes`, as well as buffers exchanged with `CryptGenRandom`. This instrumentation confirms the role of CryptoAPI in seeding OpenSSL's internal pseudo-random generator state on Windows, and allows captures to be synchronized with events observed in the provider.

In the Bitcoin 0.1.5 case, an instrumented version of the binary is used to log calls to `RAND_bytes` in `util.cpp` and to capture the values returned at the time of cryptographic

initialization. These traces relate `CryptGenRandom` outputs to application-level artifacts through the observed OpenSSL random-byte path, for example values used for key generation, without modifying the program's functional logic.

This application-level instrumentation is intentionally minimal: it alters neither the execution paths nor the internal mechanisms of the libraries used. Its role is only to provide synchronization and validation points between the application, provider, and kernel levels.

4.6 Offline Replay

Offline validations are performed in Python from the extracted dumps. The scripts do not depend on the observed Windows functions; they implement the reconstructed transformations and then compare their outputs with captured values.

A validation is considered strong when the following three conditions are met:

— the inputs are captured in the same run as the outputs;

— the offline model produces exactly the observed bytes;

— the relation is repeated across several campaigns or within a coherent full run.

This approach is used in particular to validate the transformation:

$$pool_{0x258} \longrightarrow \texttt{VeryLargeHashUpdate} \longrightarrow \texttt{seedbase\_after}.$$

In this case, the experimental proof concerns the effective consumption of the captured pool by `VeryLargeHashUpdate` and the exact reproduction of `seedbase_after`. It does not assume that every byte of the pool receives a unique semantic interpretation. Some bytes may be described as padding, metadata, or uninterpreted residues; this does not affect validation of the computation as long as the buffer actually consumed is captured and replayed bit-for-bit.

4.7 Classification of Pool Bytes

To avoid any confusion between semantic attribution and computational validation, the kernel pool is analyzed along two distinct axes.

The first axis concerns the classification of bytes in the `0x258`-byte buffer: exact copies of `ZwQuerySystemInformation` classes, transformed prefix, zeroed regions, allocation metadata, and uninterpreted residues. This classification documents the structure of the buffer and makes it possible to locate observed contributions.

The second axis concerns the reconstruction of `VeryLargeHashUpdate`. On this axis, the decisive criterion is the ability to replay offline the transformation from the captured pool to `seedbase_after`. The validated campaigns satisfy this criterion with bit-exact agreement.

This separation is important: a region may remain partially uninterpreted semantically while being fully integrated into the experimental proof of the `VeryLargeHashUpdate` transformation.

4.8 Levels of Certainty

The paper uses four status levels:

Closed transformation replayed bit-for-bit or directly observed memory relation.

Experimentally closed stable and paired relation, without complete formalization of the entire internal function.

Partially attributed observed behavior, but incomplete provenance or fine-grained semantics.

Open mechanism not sufficiently reconstructed.

These levels apply independently to each segment of the pipeline. For example, the `VeryLargeHashUpdate` transformation from captured pool to `seedbase_after` is closed, while the exhaustive semantic interpretation of all pool bytes may remain partially attributed.

Similarly, the final block $H(\texttt{state20}, \texttt{aux20})$ is closed bit-for-bit, whereas the complete upstream initialization of certain persistent provider structures remains open.

5 Global CryptGenRandom Chain

5.1 Overview

The traces show that CryptGenRandom is not merely a direct copy of a kernel buffer to the application. The output results from a composition of several stages: kernel collection, conditioning by VeryLargeHashUpdate, integration into the provider's persistent state, local transformation, final generation block, and finally copy to the user buffer.

The operational decomposition used in this study distinguishes two observed paths: a long path, including an explicit kernel collection phase, and a short path, in which no new ZwQuerySystemInformation → pool → VeryLargeHashUpdate phase is observed.

$$\begin{aligned}\textbf{Long path:}\quad & \texttt{QSI} \to \texttt{pool} \to \texttt{VLH} \to \texttt{seedbase\_after}\\ & \texttt{seedbase\_after} \to \texttt{provider persistent state} \xrightarrow{G} \texttt{state20}\\ & \texttt{local\_before} \xrightarrow{T} \texttt{aux20}\\ & (\texttt{state20}, \texttt{aux20}) \xrightarrow{H} \texttt{out40} \to \texttt{CryptGenRandom}_{\text{out}}.\end{aligned} \tag{1}$$

The short path corresponds to calls in which the provider reuses an already initialized internal state:

$$\begin{aligned}\textbf{Short path:}\quad & \texttt{CryptGenRandom} \to \texttt{provider persistent state} \to (\texttt{state20}, \texttt{aux20})\\ & \xrightarrow{H} \texttt{out40} \to \texttt{CryptGenRandom}_{\text{out}}.\end{aligned} \tag{2}$$

The Bitcoin/OpenSSL application path is distinct from this provider-side short path. It describes how CryptGenRandom can contribute to the application stack through OpenSSL's Windows-specific seeding path, while Bitcoin later consumes random bytes from OpenSSL through RAND_bytes:

$$\begin{aligned}&\texttt{Bitcoin 0.1.5} \to \texttt{OpenSSL 0.9.8h} \to \texttt{RAND\_bytes}\\ &\quad \leftarrow \texttt{OpenSSL RAND state} \leftarrow \texttt{RAND\_poll/RAND\_screen} \leftarrow \texttt{CryptGenRandom}.\end{aligned} \tag{3}$$

In this factorization, VeryLargeHashUpdate and the final block H are bit-exactly closed on validated campaigns. The residual point G denotes the internal transformation by which the provider's persistent state, fed in particular by the long path, gives rise to state20.

5.2 Short Path and Long Path

Two families of paths appear on the provider side. The short path remains confined to the provider and uses an already available internal state. The long path goes through ZwDeviceIoControlFile with IOCTL 0x00390008, then enters KSecDD to trigger kernel collection and conditioning.

The exact selection conditions between the long and short paths are not yet fully characterized. However, their existence is experimentally established. Some executions explicitly involve IOCTL 0x390008, kernel routines, and the chain ZwQuerySystemInformation → pool → VeryLargeHashUpdate → seedbase_after. Others reach the provider's final routines without any new observable kernel refresh.

Disassembly reveals a binary switching logic between these two behaviors. We denote this decision as a switching function

$$B \in \{0, 1\},$$

where $B = 1$ corresponds to the long path with an observable kernel refresh, and $B = 0$ to the short path based on reuse of the provider's persistent state. The bytes and decision site associated

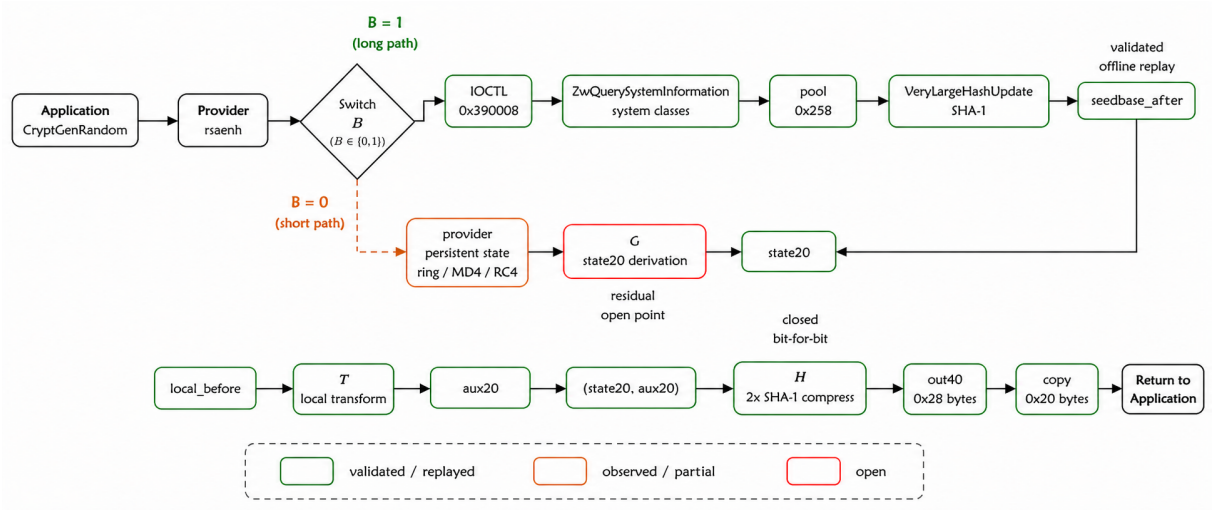

Figure 3 – Two execution paths observed in the `CryptGenRandom` pipeline. The long path ($B = 1$) includes an explicit kernel-side collection and conditioning path through IOCTL `0x390008`, `ZwQuerySystemInformation`, pool construction, and `VeryLargeHashUpdate`. The short path ($B = 0$) reuses the provider persistent state without a new observable kernel collection. The switching mechanism is identified in the disassembly, but the exact conditions governing the transition remain open.

with this switch are identified in the observed binary, but the full conditions determining its value remain open.

5.3 Global Diagram

5.4 Consequence

This structure requires the proofs to be separated. Closing `VeryLargeHashUpdate` is not sufficient to close `CryptGenRandom`, because the downstream provider path remains essential. Conversely, closing the final block H and the user copy is not sufficient to explain the complete initialization of the provider's persistent state.

The value of this study lies precisely in this decomposition. The kernel phase

$$\texttt{QSI} \to \texttt{pool} \to \texttt{VLH} \to \texttt{seedbase\_after}$$

is closed on validated campaigns. The local transformation leading to `aux20` and the final block

$$H(\texttt{state20}, \texttt{aux20})$$

are also closed. The residual lock is now localized in the upstream construction of `state20` from the provider's persistent state.

6 Kernel Chain: ZwQuerySystemInformation to Pool

6.1 Class Sequence

In the observed long paths, kernel collection follows a stable sequence of ZwQuerySystemInformation calls:

$$\texttt{0x05},\ \texttt{0x03},\ \texttt{0x07},\ \texttt{0x02},\ \texttt{0x21},\ \texttt{0x2d},\ \texttt{0x08},\ \texttt{0x17}.$$

This sequence is observed dynamically in the context of the studied CryptGenRandom execution. Campaigns capture both the individual outputs of ZwQuerySystemInformation classes and the aggregated pool subsequently passed to VeryLargeHashUpdate. The observation points are therefore paired: system calls, pool, and resulting seeds belong to the same execution path.

6.2 Aggregated Pool

The collected outputs and intermediate values populate a 0x258-byte pool. This pool is then passed to VeryLargeHashUpdate. We denote it by P:

$$|P| = \texttt{0x258} = 600.$$

The pool is then segmented into four regions of 0x96 bytes:

$$S_0 = P[\texttt{0x000} : \texttt{0x096}], \quad S_1 = P[\texttt{0x096} : \texttt{0x12c}],$$
$$S_2 = P[\texttt{0x12c} : \texttt{0x1c2}], \quad S_3 = P[\texttt{0x1c2} : \texttt{0x258}].$$

This segmentation is not an abstract assumption: it corresponds to buffers actually captured at the entry of VeryLargeHashUpdate and used in offline replay.

6.3 Class Attribution

The campaigns distinguish two levels of analysis.

The first level is the semantic attribution of pool bytes to observed ZwQuerySystemInformation classes. Several classes appear as contiguous and stable copies:

— class 0x03: exact copy observed at offset 0x50;
— class 0x07: exact copy observed around 0x88;
— class 0x02: large exact copy observed starting at 0xa8;
— class 0x21: exact copy observed around 0x1e8;
— class 0x2d: exact copy observed around 0x200.

Other regions of the pool correspond to prefixes, padding, metadata, or residual values whose precise semantics cannot always be attributed to a single ZwQuerySystemInformation class. This is notably the case for some contributions associated with classes 0x05, 0x08, and 0x17, as well as for structural or alignment regions.

The second level, more important for this study, is computational: independently of the semantic labeling of each byte, the full pool P is captured and its consumption by VeryLargeHashUpdate is replayed offline. At this level, the reconstruction is complete for validated campaigns.

6.4 Coverage and Interpretation

Initial attribution campaigns yielded a direct coverage of approximately 0x1b0 bytes, or about 72% of the pool, when only contiguous and immediately identifiable ZwQuerySystemInformation segments were considered.

This figure should not be interpreted as the coverage of the kernel reconstruction. It only reflects a conservative measure of direct semantic attribution. Subsequent campaigns show that the captured pool is fully available and that the transformation:

$$P \to (S_0, S_1, S_2, S_3) \to \texttt{VeryLargeHashUpdate} \to \texttt{seedbase\_after}$$

is validated bit-for-bit. In other words, the semantic labeling of some bytes remains incomplete, but the computational reconstruction of `VeryLargeHashUpdate` from the captured pool is complete.

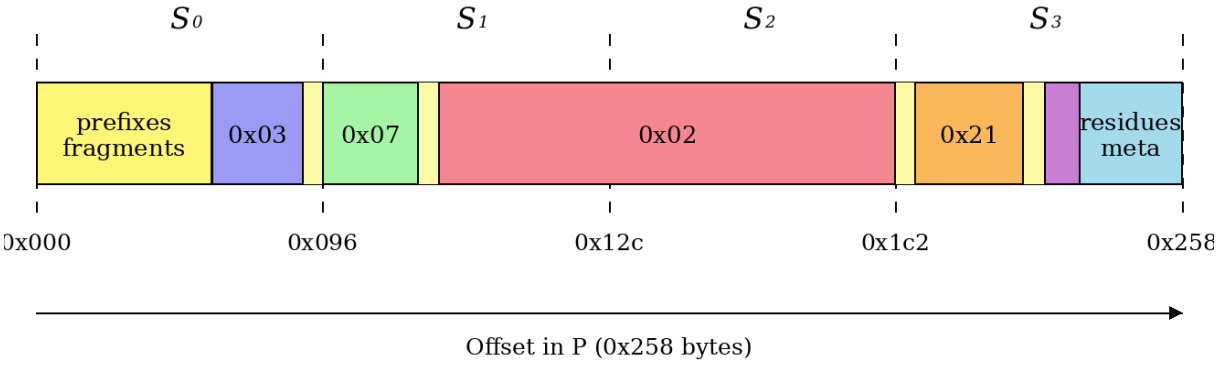

Figure 4 – **Complete pool captured and consumed by `VeryLargeHashUpdate`.** Experimental view of the pool P. Some regions are directly attributed to contiguous `ZwQuerySystemInformation` classes; others correspond to fragments, padding, metadata, or residual data. Dashed lines indicate the effective segmentation into four `0x96`-byte blocks. For `VeryLargeHashUpdate` replay, the key point is that the complete pool is captured and consumed bit-for-bit.

6.5 Status

The status of this component is therefore as follows. The `ZwQuerySystemInformation` collection sequence is observed dynamically. The structure of the `0x258`-byte pool and its segmentation into four `0x96`-byte blocks are established experimentally. The fine-grained semantic attribution of each pool byte to system classes remains partially descriptive for certain regions.

From the perspective of reconstructing the kernel chain, however, the essential result is closed: for validated campaigns, the captured pool allows exact replay of `VeryLargeHashUpdate` and bit-for-bit reproduction of `seedbase_after`. The next section formalizes this transformation.

7 Reconstruction and Validation of `VeryLargeHashUpdate`

7.1 Identification of SHA-1

Static and dynamic analysis of `VeryLargeHashUpdate` reveals the use of a SHA-1 implementation in `KSecDD`. The observed initialization constants correspond to the standard SHA-1 initialization vector:

```
mov [eax+18h], 67452301h
mov [eax+1Ch], 0EFCDAB89h
mov [eax+20h], 98BADCFEh
mov [eax+24h], 10325476h
mov [eax+28h], 0C3D2E1F0h
```

The traces show repeated sequences of the form `SHA1_Init`, `SHA1_Update`, and `SHA1_Final`. In the campaigns, these calls are observed around buffers corresponding to the captured pool and to `seedbase` regions. The WinDbg scripts explicitly extract the pool, the four segments S_i, as well as `seedbase_before` and `seedbase_after` at the entry and exit of `VeryLargeHashUpdate`.

7.2 Input Structure

The pool P passed to `VeryLargeHashUpdate` has size `0x258` bytes. It is split into four `0x96`-byte segments:

$$S_0 = P[\texttt{0x000} : \texttt{0x096}], \quad S_1 = P[\texttt{0x096} : \texttt{0x12c}],$$
$$S_2 = P[\texttt{0x12c} : \texttt{0x1c2}], \quad S_3 = P[\texttt{0x1c2} : \texttt{0x258}].$$

The input seeds are denoted:

$$\texttt{seed}_0,\ \texttt{seed}_1,\ \texttt{seed}_2,\ \texttt{seed}_3.$$

Each seed occupies `20` bytes. The complete `seedbase` state therefore occupies:

$$4 \times 20 = 80 = \texttt{0x50} \text{ bytes.}$$

7.3 First Phase

The traces and offline replay are consistent with a first phase producing four intermediate `20`-byte digests:

$$L_0 = \mathrm{SHA1}(\texttt{seed}_0 \parallel S_0 \parallel \texttt{seed}_1 \parallel S_1), \tag{4}$$
$$L_1 = \mathrm{SHA1}(\texttt{seed}_1 \parallel S_1 \parallel \texttt{seed}_0 \parallel S_0), \tag{5}$$
$$L_2 = \mathrm{SHA1}(\texttt{seed}_2 \parallel S_2 \parallel \texttt{seed}_3 \parallel S_3), \tag{6}$$
$$L_3 = \mathrm{SHA1}(\texttt{seed}_3 \parallel S_3 \parallel \texttt{seed}_2 \parallel S_2). \tag{7}$$

These intermediate values are then reused in a second phase. The important point is not merely the identification of SHA-1, but the exact correspondence between this model and the values observed in the campaigns.

7.4 Second Phase

The second phase operates on the intermediate digests:

$$\texttt{seed}'_0 = \mathrm{SHA1}(L_0 \parallel L_2), \tag{8}$$
$$\texttt{seed}'_1 = \mathrm{SHA1}(L_1 \parallel L_3), \tag{9}$$
$$\texttt{seed}'_2 = \mathrm{SHA1}(L_2 \parallel L_0), \tag{10}$$
$$\texttt{seed}'_3 = \mathrm{SHA1}(L_3 \parallel L_1). \tag{11}$$

The concatenated result constitutes `seedbase_after`:

$$\texttt{seedbase\_after} = \texttt{seed}'_0 \parallel \texttt{seed}'_1 \parallel \texttt{seed}'_2 \parallel \texttt{seed}'_3.$$

7.5 Offline Validation

The validation scripts take the following dumps as input:

— `pool.bin`;
— `s0.bin`, `s1.bin`, `s2.bin`, `s3.bin`;
— `seedbase_before.bin`;
— `seedbase_after.bin`;
— the four files `seedN_before.bin` and `seedN_after.bin`.

From these dumps, the replay reconstructs the four values L_i, then the four updated seeds, and compares the result against the values captured in memory.

In validated campaigns, the four updated seeds and `seedbase_after` exactly match the observed values. A representative output is:

```
$ python vlh_campaign_report.py camp01 ... camp05

[GLOBAL]
  campaigns                   : 5
  pool coverage               : 0x258 / 0x258 (100%)
  covered differing bytes     : 0
  uncovered ranges            : none

[VLH REPLAY]
  pool segmentation           : 4 x 0x96-byte blocks
  seed0prime                  : bitwise identical
  seed1prime                  : bitwise identical
  seed2prime                  : bitwise identical
  seed3prime                  : bitwise identical
  seedbase_after              : bitwise identical

  overall verdict             : full bitwise agreement
```

Recent campaigns confirm this result across several runs: the computed values for `seed0prime` through `seed3prime`, as well as `seedbase_after`, match the observed values, with `PASS = True`. This establishes the computational closure of the stage:

$$P \to \texttt{VeryLargeHashUpdate} \to \texttt{seedbase\_after}.$$

7.6 State Continuity

The observations are consistent with a state transition: `VeryLargeHashUpdate` does not merely produce a temporary output, but updates a `0x50`-byte `seedbase` region. This region is captured before and after the call, making it possible to treat `VeryLargeHashUpdate` as a transition function:

$$\texttt{seedbase\_after} = F_{\texttt{VeryLargeHashUpdate}}(\texttt{seedbase\_before}, P).$$

Within the scope of validated campaigns, this transition function is replayed bit-for-bit. If chained runs are used, continuity between a `seedbase_after` output and a subsequent `seedbase_before` input must be documented separately, so as not to conflate closure of an isolated call with a complete proof of inter-call persistence.

7.7 Status

The transformation:

$$\texttt{seedbase\_before}, P \rightarrow \texttt{VeryLargeHashUpdate} \rightarrow \texttt{seedbase\_after}$$

is considered closed within the scope of the validated campaigns. Offline replay reproduces exactly the four updated seeds and the concatenated `seedbase_after`.

The remaining uncertainty regarding the fine-grained provenance of certain portions of the pool does not affect this validation. It concerns the preceding stage, namely the complete semantic attribution of all bytes of P to `ZwQuerySystemInformation` classes, metadata, padding regions, or residues. For `VeryLargeHashUpdate` itself, the computation from the captured pool is closed.

8 `rsaenh` Provider: State, Primitives, and Internal Paths

8.1 Role of the Provider

After the return from the kernel path, the `rsaenh` provider retains an active role. It does not directly pass to the caller a buffer produced by `KSecDD`. Instead, traces show additional operations involving cryptographic primitives, local buffers, persistent provider state, a circular buffer, a final generation block, and a final copy to the caller.

In the observed long paths, the call to `ZwDeviceIoControlFile` with IOCTL `0x00390008` provides a `0x100`-byte *OutputBuffer*. In the paired runs studied, this buffer is reused by the provider as key material for RC4 initialization. It is therefore not the final output of `CryptGenRandom`, but an internal input to the provider path.

8.2 Observed Primitives

Several families of routines are observed in the provider:

— MD4: initialization, update, finalization, and compression;
— SHA-1: initialization, update, and finalization routines;
— SHA-1 compression on `64`-byte blocks in the final block;
— RC4: KSA, PRGA, and XOR;
— XOR operations and memory copies;
— a `0x100`-byte circular buffer advancing in steps of `0x10`.

These primitives are identified by disassembly and dynamic traces. Some are also locally validated through offline replay: for captured inputs, the outputs computed in Python match the values observed in memory. This validation establishes the effective role of the primitives in the studied paths, without claiming by itself to close the full temporal composition of the provider state.

8.3 MD4 and State Update

A representative example concerns a standard MD4 computation over a captured `20`-byte input buffer. The digest replayed offline matches the digest observed after finalization, and then the `16` bytes injected into the provider state.

In the studied run, the destination state before injection was zero, making the relation particularly clear:

$$\text{state}_{after} = \text{state}_{before} \oplus \text{MD4}(X).$$

When:

$$\text{state}_{before} = 0,$$

the observed operation is equivalent to copying the MD4 digest into the destination region.

This validation is local. It confirms that MD4 effectively participates in a provider-state update, but it is not sufficient to close the global mapping:

$$\texttt{seedbase\_after} \longrightarrow \text{provider\_state} \longrightarrow \texttt{state20}.$$

8.4 Provider SHA-1 and `64`-Byte Compressions

The provider also contains SHA-1 routines distinct from those observed in `KSecDD` for `VeryLargeHashUpdate`. The campaign observations include points corresponding to SHA-1 initialization, update, and finalization.

Particular attention is given to the final block. Unlike `VeryLargeHashUpdate`, which uses full `SHA1_Init`/`SHA1_Update`/`SHA1_Final`-style cycles, the final block manipulates `64`-byte blocks and applies SHA-1 compressions in a specific construction. This part is treated separately in the section devoted to H, since it does not correspond to a simple SHA-1 invocation over an arbitrary message with standard padding and finalization.

8.5 RC4

The observed RC4 routines correspond to classical KSA and PRGA structures. At the observed KSA entry, one register points to the RC4 permutation S, initialized as:

$$S = (0, 1, 2, \ldots, 255),$$

while another points to the `0x100`-byte key material.

In the studied run, the return from IOCTL `0x00390008` is observed with:

$$\text{InputLength} = \texttt{0x100},$$
$$\text{OutputLength} = \texttt{0x100}.$$

After the IOCTL returns, the `0x100`-byte output buffer contains the bytes returned by the `KSecDD` path. At the beginning of the RC4 KSA, the same buffer is used as key material. The memory dumps therefore show, within the same run, the identity:

$$K_{RC4} = \text{OutputBuffer}_{IOCTL}.$$

In other words, the key material used by the `rsaenh` RC4 KSA is the buffer returned by the `KSecDD` path through IOCTL `0x00390008`.

This observation locally closes the link:

$$\text{IOCTL}_{0x00390008}\ \text{OutputBuffer} \longrightarrow K_{RC4} \longrightarrow S_{after}.$$

It does not yet, by itself, demonstrate the full causality:

$$\texttt{seedbase\_after} \longrightarrow K_{RC4}.$$

To establish this latter relation strongly, `seedbase_after` and K_{RC4} must be captured in the same run, and a controlled modification of `seedbase_after` must be shown to propagate to the buffer returned by the IOCTL and, consequently, to the state S_{after}.

8.6 Circular Buffer

The traces support the existence of a `0x100`-byte circular buffer in the provider. Digests or intermediate blocks are injected into it by XOR, with a cursor advancing in steps of `0x10`. This structure explains why global provider reconstruction is more difficult than local validation of individual primitives.

Even when MD4, RC4, or SHA-1 are identified and locally validated, their temporal composition depends on the call order, circular-buffer cursor, persistent state, and the selection between the short and long paths. The global mapping to `state20` therefore requires capturing the relevant writes in the effective execution order.

8.7 IOCTL and Non-Identity with the Output

IOCTL `0x00390008` is observed on the long path. The traces show that its `0x100`-byte *OutputBuffer* is reused as key material for the RC4 KSA in `rsaenh`.

However, the output returned by `CryptGenRandom` is not a direct prefix of the IOCTL buffer. The IOCTL buffer feeds an internal provider stage, while the user output comes from a provider

buffer produced after several additional transformations: state update, local operations, preparation of `state20` and `aux20`, final generation block, and then copy of the first `0x20` bytes to the caller.

The path must therefore be separated as follows:

$$\texttt{kernel/IOCTL} \to \texttt{provider (RC4/state)} \to (\texttt{state20}, \texttt{aux20}) \to \texttt{out40} \to \texttt{CryptGenRandom}_{out}.$$

8.8 Status

The status of this part is mixed. The MD4, RC4, and SHA-1 primitives are identified, and some local relations are closed by replay or direct memory identity. In particular, the link between the *OutputBuffer* of IOCTL `0x00390008` and the RC4 KSA key material is established in paired runs.

However, the complete composition of the provider state remains partially open. The residual lock does not concern the existence of the primitives, nor the final output copy, but the complete reconstruction of the transition:

$$\texttt{seedbase\_after} \longrightarrow \text{provider\_state} \longrightarrow \texttt{state20}.$$

9 Provider Bridge to the Final Block: G, T, and H

9.1 Functional Decomposition

The last part of the pipeline can be factorized as follows:

$$\texttt{seedbase\_after}/\text{provider\_state} \xrightarrow{G} \texttt{state20}$$

$$\texttt{local\_before} \xrightarrow{T} \texttt{aux20}$$

$$(\texttt{state20}, \texttt{aux20}) \xrightarrow{H} \texttt{out40}.$$

This factorization separates three distinct problems. G concerns the provider's persistent state and the preparation of `state20`. T concerns the local branch leading to `aux20`. H concerns the final generation block.

This separation is methodologically useful: it avoids conflating closure of the final block with complete reconstruction of the provider state. In the current campaigns, H is closed bit-for-bit, T is experimentally closed at the level of its output relevant to H, and G is reduced to a more precise upstream sub-problem.

9.2 Experimental Closure of T

Provider-level traces show that the local buffer observed at the `AFTER_LOCAL` stage is reused unchanged at the entry of the final block. In paired executions, we observe:

$$\texttt{AFTER\_LOCAL} = \texttt{local20} = \texttt{aux20}.$$

Thus, in the runs considered, no additional transformation of the local buffer is observed between the end of the local branch and the entry of the final block.

This closure must be understood precisely. It does not mean that the entire internal function T is mathematically formalized. It means that its result relevant to the final block is captured, stabilized, and identical to `aux20` at the time of entry into the final block.

9.3 Projection of `state20` into the Global Slot

Recent campaigns show that `state20` is copied into the provider global slot before entry into the final block. The observed write corresponds to a copy of five 32-bit words, i.e. `0x14` bytes:

```
; Representative state20 copy
; copies 20 bytes from the local source buffer
; into the provider global slot

mov esi, <local_source>
mov edi, <provider_state20_slot>
mov ecx, 5
rep movsd
```

This observation establishes that `state20` is not computed by the final block. The final block consumes a 20-byte value already prepared and placed in the provider global slot.

The observed relation is therefore:

$$\texttt{source20} \longrightarrow \texttt{provider\_state20\_slot} \longrightarrow \texttt{state20}.$$

9.4 Immediate Source of `state20`

Stepping one level upstream shows that the local source of this copy is itself populated from a provider global structure. In the traces studied, the observed terminal relation is:

$$\texttt{provider\_source\_structure} \longrightarrow [\texttt{ebp} - \texttt{0x28}] \longrightarrow \texttt{provider\_state20\_slot}.$$

One observed code site contains an immediate reference to this provider source structure, then copies `0x14` bytes to a local region before calling the provider orchestration routine. The latter then orchestrates the projection into the global slot used by the final block.

This observation is important because it shifts the experimental lock. The question is no longer whether `state20` is produced inside the final block: it is not. The question becomes: how is the persistent provider structure used as the source initialized and updated before this copy?

9.5 Reduction of G

Before these observations, G could be broadly formulated as:

$$\texttt{seedbase\_after}/\text{provider\_state} \longrightarrow \texttt{state20}.$$

After the recent campaigns, the terminal part of G can be restricted to:

$$\texttt{provider source structure} \longrightarrow \texttt{source20} \longrightarrow \texttt{provider\_state20\_slot} \longrightarrow \texttt{state20}.$$

The remaining lock therefore no longer concerns the projection of `state20` into the final block, but the upstream construction of the source structure.

Within the inspected disassembly range, one immediate reference to this source structure was identified. However, it remains possible that other writes reach this region indirectly, through a pointer, structure initialization, or copy from another memory region. This possibility must remain open until writes to this structure have been exhaustively captured.

9.6 Status

The status of this part is as follows. T is experimentally closed at the level of its useful result:

$$\texttt{AFTER\_LOCAL} = \texttt{aux20}.$$

The terminal projection of G into `state20` is observed:

$$\texttt{provider\_source\_structure} \longrightarrow [\texttt{ebp} - \texttt{0x28}] \longrightarrow \texttt{provider\_state20\_slot} \longrightarrow \texttt{state20}.$$

The final block H is closed bit-for-bit in the next section. The remaining open point therefore concerns the upstream maintenance of the persistent provider source structure, and not the final generation nor the copy of `state20` into the global slot.

10 Final Block H: SHA-1-Based Generation

10.1 Inputs

The final block consumes two useful 0x14-byte buffers: state20 and aux20.

The value state20 comes from the provider global slot. It is prepared before entry into the final block. The value aux20 comes from the provider's local branch and corresponds to the buffer observed at the AFTER_LOCAL stage.

The final block therefore does not directly collect entropy. It consumes two provider values that have already been stabilized.

10.2 Computation Structure

The observations and offline replay lead to the following model. The computation produces two 0x14-byte parts, denoted part0 and part1. Their concatenation forms out40:

$$\texttt{out40} = \texttt{part0} \parallel \texttt{part1}.$$

The validated abstract scheme is:

$$\begin{aligned}
\texttt{tmp0} &= \mathrm{add}_{160}^{BE}(\texttt{state20}, \texttt{aux20}), && (12)\\
\texttt{x0} &= \mathrm{SHA1Compress}_{IV}(\texttt{tmp0} \parallel 0^{44}), && (13)\\
\texttt{part0} &= \mathrm{add}_{160}^{BE}(\texttt{state20}, \texttt{x0}, 1), && (14)\\
\texttt{tmp1} &= \mathrm{add}_{160}^{BE}(\texttt{part0}, \texttt{aux20}), && (15)\\
\texttt{x1} &= \mathrm{SHA1Compress}_{IV}(\texttt{tmp1} \parallel 0^{44}), && (16)\\
\texttt{part1} &= \mathrm{add}_{160}^{BE}(\texttt{part0}, \texttt{x1}, 1), && (17)\\
\texttt{out40} &= \texttt{part0} \parallel \texttt{part1}. && (18)
\end{aligned}$$

The additions are interpreted as additions over 160-bit integers in big-endian order. The final parameter 1 in the additions producing part0 and part1 corresponds to the initial carry observed in the validated model.

The primitive used is a SHA-1 compression over a constructed 64-byte block:

$$\texttt{tmp} \parallel 0^{44}.$$

Here, 0^{44} denotes 44 zero bytes. This is therefore not a complete SHA-1 call with standard padding, length encoding, and finalization over a 20-byte message. The compression block is constructed explicitly from a 20-byte value followed by 44 zero bytes.

10.3 Offline Validation

The offline validator reconstructs the 64-byte blocks consumed by the final block from only the captured inputs state20 and aux20. In the representative run below, the state20 input is:

```
state20_entry:
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14
```

The first reconstructed block begins with these 20 bytes and is padded with zeros:

```
p1_block64:
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
```

The observed execution produces the following buffer:

```
out40_after:
0a e8 1e 95 fc 39 7b 50 8d 9a 4e 0d fd cd 1a 2c
24 e4 72 86 18 da e5 26 43 1b ff 1f d7 24 c3 6b
63 32 cc 84 46 91 7f 4f
```

The independent replay reproduces exactly these 40 bytes:

```
replay_std:
0a e8 1e 95 fc 39 7b 50 8d 9a 4e 0d fd cd 1a 2c
24 e4 72 86 18 da e5 26 43 1b ff 1f d7 24 c3 6b
63 32 cc 84 46 91 7f 4f

compress_std replay == out40_after : True
compress_ns  replay == out40_after : False
```

The comparison also distinguishes the correct compression mode: the standard mode reproduces the execution, whereas the NS variant does not. The test therefore validates not only output equality, but also discriminates between two candidate models.

The final copy is verified separately:

```
final_dst20_after == out40_after[:0x20] : True
```

Thus, for this run, the closed relation is:

$$H(\texttt{state20}, \texttt{aux20}) = \texttt{out40}, \qquad \texttt{CryptGenRandom}_{out} = \texttt{out40}[0 : \texttt{0x20}].$$

The sequence `01 02 03 ...` reflects a controlled initialization used for validation. Identical replay results are obtained with non-trivial values observed in normal operation, indicating that the reconstruction is not specific to this simplified case.

10.4 Status

The final block is considered bit-exactly closed within the scope of the validated campaigns. The inputs `state20` and `aux20` are captured at entry into the final block, the offline computation reproduces `out40`, and the `0x28` bytes produced match the buffer observed after execution.

This closure removes the final block from the list of major uncertainties. Even though the upstream construction of `state20` remains open, final generation from `state20` and `aux20` is deterministic, captured, and replayable.

11 Final Copy to the User Buffer

11.1 Observation of the Copy

The last observable stage of the pipeline is a memory copy from a provider buffer to the buffer supplied by the caller of `CryptGenRandom`. It is observed as a `rep movs`-style instruction.

At the time of this copy, the registers show that the source points to the provider buffer containing `out40`, while the destination points to the user buffer supplied to `CryptGenRandom`. The copied length corresponds to the number of bytes requested by the caller. In the runs studied, this length is `0x20` bytes.

11.2 Content Validation

Direct comparison shows that, for the observed calls requesting `0x20` bytes, the source of the copy corresponds to the prefix of `out40`:

$$\texttt{final\_src32} = \texttt{out40}[\texttt{0x00} : \texttt{0x20}].$$

After execution of the copy, the destination buffer contains the same value:

$$\texttt{final\_dst32\_after} = \texttt{out40}[\texttt{0x00} : \texttt{0x20}].$$

Thus, in the validated `0x20`-byte output runs:

$$\texttt{CryptGenRandom}_{out} = \texttt{out40}[\texttt{0x00} : \texttt{0x20}].$$

The interval $[\texttt{0x00} : \texttt{0x20}]$ here denotes the first `0x20` bytes, that is, `32` bytes, of the `0x28`-byte `out40` buffer.

11.3 Consequence

This observation closes the link between the final block and the output visible to the application for the `0x20`-byte runs considered. It confirms that the output of `CryptGenRandom` is not a direct prefix of the kernel buffer returned by IOCTL `0x390008`. The IOCTL buffer feeds an internal provider stage, whereas the user output comes from the `out40` buffer produced by the final block and then copied to the caller.

In the studied bit-exact replay runs, the caller-requested output length is `0x20` bytes. Additional instrumentation campaigns with larger requested output sizes (`0x40`, `0x80`, and `0x100`) confirm the copy-side mechanism: the number of bytes copied to the caller-supplied buffer follows the requested length, and the final transfer still uses the same provider-to-user copy pattern. These larger-output campaigns are used only to validate the copy-length behavior. They do not extend the bit-exact `out40` reconstruction beyond the `0x20`-byte output cases analyzed above.

12 Link Between OpenSSL 0.9.8h and Bitcoin 0.1.5

12.1 OpenSSL on Windows

On Windows, OpenSSL 0.9.8h uses several system sources in `RAND_poll` to seed its internal pseudo-random generation state. Among these sources, CryptoAPI plays a specific role: OpenSSL attempts to acquire a Windows cryptographic context, then calls `CryptGenRandom` to obtain bytes produced by the system cryptographic provider.

The relevant CryptoAPI path is located in OpenSSL's Windows-specific RAND implementation, rather than directly in the Bitcoin source tree. It can be summarized by the following excerpt:

```
HCRYPTPROV hProvider = 0;
BYTE buf[64];

if (CryptAcquireContext(&hProvider, NULL, NULL,
                        PROV_RSA_FULL,
                        CRYPT_VERIFYCONTEXT)) {
    if (CryptGenRandom(hProvider, sizeof(buf), buf))
        RAND_add(buf, sizeof(buf), 0);
    CryptReleaseContext(hProvider, 0);
}
```

This excerpt illustrates the seeding of OpenSSL's internal RAND state via `CryptGenRandom`, called from `RAND_poll` on Windows. This representation is not intended to reproduce the entirety of `RAND_poll` verbatim. It isolates the experimentally relevant link: on Windows, OpenSSL may incorporate into its internal state bytes obtained through `CryptGenRandom`.

The entropy estimate passed to `RAND_add` belongs to OpenSSL's internal accounting. In the OpenSSL 0.9.8h code path shown above, the `CryptGenRandom` bytes are added with an entropy estimate of zero, even though they are still mixed into the RAND state. This estimate is therefore not a direct experimental measurement of the effective entropy of the bytes returned by `CryptGenRandom`. In this study, observed values are analyzed as measurable contributions and states, not as quantitative guarantees of cryptographic entropy.

12.2 Additional OpenSSL Sources in `RAND_poll`

The CryptoAPI call is only one component of OpenSSL's Windows collection routine. The same `RAND_poll` implementation also incorporates other system-derived values, including network statistics, window and cursor state, message queue state, Toolhelp32 enumeration data, timing information, memory status, and the current process identifier.

In simplified form, the OpenSSL Windows collection path includes sources of the following kind:

— `NetStatisticsGet` for `LanmanWorkstation` and `LanmanServer`;
— `CryptGenRandom` through CryptoAPI;
— `GetForegroundWindow`, `GetCursorInfo`, and `GetQueueStatus`;
— `Heap32List`, `Heap32Entry`, `ProcessEntry`, `ThreadEntry`, and `ModuleEntry`;
— `QueryPerformanceCounter` or related timer sources;
— `GlobalMemoryStatus`;
— `GetCurrentProcessId`.

These sources belong to OpenSSL's user-space collection layer. They must not be confused with the `ZwQuerySystemInformation` classes observed inside the Windows kernel path triggered by `CryptGenRandom`. The two mechanisms operate at different levels: OpenSSL aggregates user-space system sources into its own RAND state, while `CryptGenRandom` internally invokes the Windows provider and, on the long path, kernel collection and conditioning.

12.3 Bitcoin 0.1.5 Use of OpenSSL

Bitcoin 0.1.5 depends on OpenSSL for its cryptographic operations. The historical Bitcoin 0.1.5 build instructions explicitly mention OpenSSL 0.9.8h as a dependency, alongside wxWidgets, Berkeley DB, and Boost.

The Bitcoin source does not itself contain the OpenSSL `RAND_poll` CryptoAPI code shown above. Instead, Bitcoin calls OpenSSL's random generation interfaces and also contributes application-level seed material to OpenSSL's RAND state.

In `util.cpp`, Bitcoin initializes OpenSSL locking support and then seeds the OpenSSL random generator during static initialization:

```
class CInit
{
public:
    CInit()
    {
        // Init openssl library multithreading support
        lock_cs = (HANDLE*)OPENSSL_malloc(CRYPTO_num_locks() *
                                          sizeof(HANDLE));
        for (int i = 0; i < CRYPTO_num_locks(); i++)
            lock_cs[i] = CreateMutex(NULL,FALSE,NULL);
        CRYPTO_set_locking_callback(win32_locking_callback);

        // Seed random number generator with screen scrape
        // and other hardware sources
        RAND_screen();

        // Seed random number generator with perfmon data
        RandAddSeed(true);
    }
};
```

The call to `RAND_screen()` is an OpenSSL call. In the Windows OpenSSL implementation, `RAND_screen()` invokes `RAND_poll()` and then performs additional screen-based seeding. Therefore, in the studied Windows configuration, Bitcoin's initialization can trigger OpenSSL's Windows RAND collection path, including the CryptoAPI/`CryptGenRandom` component.

Bitcoin also defines its own `RandAddSeed` routine, which contributes additional application-level material to OpenSSL's RAND state:

```
void RandAddSeed(bool fPerfmon)
{
    // Seed with CPU performance counter
    LARGE_INTEGER PerformanceCount;
    QueryPerformanceCounter(&PerformanceCount);
    RAND_add(&PerformanceCount, sizeof(PerformanceCount), 1.5);
    memset(&PerformanceCount, 0, sizeof(PerformanceCount));

    static int64 nLastPerfmon;
    if (fPerfmon || GetTime() > nLastPerfmon + 5 * 60)
    {
        nLastPerfmon = GetTime();

        // Seed with the entire set of perfmon data
        unsigned char pdata[250000];
        memset(pdata, 0, sizeof(pdata));
        unsigned long nSize = sizeof(pdata);
        long ret = RegQueryValueEx(HKEY_PERFORMANCE_DATA,
                                   "Global", NULL, NULL,
                                   pdata, &nSize);
        RegCloseKey(HKEY_PERFORMANCE_DATA);
        if (ret == ERROR_SUCCESS)
        {
            uint256 hash;
            SHA256(pdata, nSize, (unsigned char*)&hash);
            RAND_add(&hash, sizeof(hash),
                     min(nSize/500.0, (double)sizeof(hash)));
            hash = 0;
            memset(pdata, 0, nSize);
        }
    }
}
```

This Bitcoin-level seeding is distinct from OpenSSL's own `RAND_poll` collection. It adds a performance counter and, when available, a SHA-256 digest of Windows performance data to the OpenSSL RAND state. Thus, Bitcoin 0.1.5 does not rely solely on `CryptGenRandom`; rather, it uses OpenSSL's RAND machinery, which itself may include `CryptGenRandom` on Windows, and adds further seed material from Bitcoin's own initialization code.

The build file also indicates that Bitcoin does not use OpenSSL's encryption routines and proposes a `no-everything` build intended to retain only the necessary components, in particular the `libeay32` library. This point is important: the OpenSSL dependency does not mean that Bitcoin uses all primitives provided by OpenSSL, but it does establish that the client depends on this library for part of its cryptographic path.

12.4 Random Bytes Requested by Bitcoin

Bitcoin 0.1.5 obtains random values through OpenSSL calls such as `RAND_bytes`. A representative example is the `GetRand` routine:

```
uint64 GetRand(uint64 nMax)
{
    if (nMax == 0)
        return 0;

    uint64 nRange = (_UI64_MAX / nMax) * nMax;
    uint64 nRand = 0;
    do
        RAND_bytes((unsigned char*)&nRand, sizeof(nRand));
    while (nRand >= nRange);
    return (nRand % nMax);
}
```

This code shows Bitcoin consuming bytes from OpenSSL's RAND state through `RAND_bytes`. It does not call `CryptGenRandom` directly. The connection to `CryptGenRandom` is indirect and passes through OpenSSL's Windows-specific RAND collection logic.

The relevant historical relationship is therefore:

$$\text{Bitcoin 0.1.5} \xrightarrow{\text{calls}} \texttt{RAND\_bytes} \xleftarrow{\text{draws from}} \text{OpenSSL RAND state.}$$

That OpenSSL RAND state may have previously been seeded through:

$$\texttt{RAND\_poll/RAND\_screen} \rightarrow \texttt{CryptGenRandom} \rightarrow \text{Windows XP RNG,}$$

as well as through other OpenSSL and Bitcoin-level sources such as `RandAddSeed`. This formulation is important: `CryptGenRandom` is not the only source influencing OpenSSL's internal state, and `RAND_bytes` does not imply a direct call to `CryptGenRandom` at each use. It establishes, however, that in a historical Windows deployment of Bitcoin 0.1.5 using OpenSSL 0.9.8h, the Windows XP RNG can contribute to the OpenSSL state from which Bitcoin later draws random bytes.

12.5 Experimental Validation of the Path

In the experimental environment studied, traces show that executing Bitcoin 0.1.5 on Windows reaches OpenSSL's random generation path and that OpenSSL's Windows RAND implementation can invoke `CryptGenRandom` as part of its collection. This point is distinct from the mere build-time dependency on OpenSSL: it is a dynamic observation of the path actually taken.

The established link is therefore threefold:

— at the software level, Bitcoin 0.1.5 depends on OpenSSL 0.9.8h;

— at initialization level, Bitcoin calls OpenSSL seeding routines such as `RAND_screen()` and adds further seed material through `RandAddSeed`;

— at execution level, OpenSSL on Windows may call `CryptGenRandom` as part of `RAND_poll`.

Thus, the properties of the Windows XP RNG may contribute to OpenSSL's internal state in a historical Windows deployment of Bitcoin 0.1.5. This observation justifies the relevance of an empirical reconstruction of `CryptGenRandom` on Windows XP SP3, not as direct evidence of exploitation, but as an analysis of a system component effectively present in the historical generation chain.

12.6 System Sources Observed in `RAND_poll`

Captures of the OpenSSL/RAND layer document the system sources aggregated in the experimental environment. The main campaign consists of 50 independent executions. The

`randwin_data.bin` files were reconstructed into `randwin_full.json` representations, then compared using *source-occurrence* alignment, that is, by source name and rank of appearance during collection.

The observed sources include, in particular:

— `LanmanWorkstation`
— `LanmanServer`
— `CryptGenRandom`
— `GetForegroundWindow`
— `GetCursorInfo`
— `GetQueueStatus`
— `Heap32List`
— `Heap32Entry`
— `ProcessEntry`
— `ThreadEntry`
— `ModuleEntry`
— `QueryPerformanceCounter`
— `GlobalMemoryStatus`
— `GetCurrentProcessId`

These sources belong to OpenSSL's user-space collection. They must not be confused with the `ZwQuerySystemInformation` classes observed in the internal Windows path triggered by `CryptGenRandom`.

12.7 Post-Stir Replay of `ssleay_rand_bytes`

The Windows sources described above characterize the data collected by OpenSSL's Windows `RAND_poll` layer. They are not modeled here as a simple concatenation of byte strings. Instead, the experimental separation is as follows: the `randwin` traces validate the observed Windows-side records, while the `ssleay` replay validates the byte-generation loop after OpenSSL's internal state has already been stirred.

Let

$$R_i = (\text{source}_i, \text{len}_i, \text{bytes}_i)$$

denote a Windows record collected during `RAND_poll`. The `randwin_full.json` representation records these objects and allows structural validation of selected Windows data structures, such as `LanmanWorkstation`, `LanmanServer`, Toolhelp32 enumeration records, timing values, memory status records, and `CryptGenRandom` output blocks. This validation does not by itself describe the complete OpenSSL state transition induced by `RAND_poll`.

The replay boundary used in this work is the post-stir OpenSSL state:

$$S = \texttt{state\_after\_stir}, \qquad p = \texttt{state\_index\_after\_stir}, \qquad N = \texttt{state\_num\_after\_stir}.$$

In the representative trace, these values are:

$$p = 240, \qquad N = 1023.$$

For a 32-byte request to `ssleay_rand_bytes`, the observed loop uses four iterations. At iteration j, the replay extracts a 10-byte state window

$$s_j = S[p + 10j : p + 10(j + 1)].$$

The digest for the iteration is recomputed independently as

$$d_j = \mathrm{SHA1}\left(\texttt{local\_md}_j \,\|\, \texttt{md\_c}_j \,\|\, \texttt{input\_buf}_j \,\|\, s_j\right).$$

The first half of the digest updates the OpenSSL state window by XOR:

$$S[p + 10j : p + 10(j + 1)] \leftarrow s_j \oplus d_j[0 : 10],$$

while the second half contributes output bytes:

$$\mathrm{out}_j = d_j[10 : 20].$$

For a 32-byte output, the concatenation is truncated in the final iteration:

$$\texttt{RAND\_bytes}[0 : 32] = d_0[10 : 20] \,\|\, d_1[10 : 20] \,\|\, d_2[10 : 20] \,\|\, d_3[10 : 12].$$

For the first iteration of the representative trace, the selected state window is

$$s_0 = \texttt{9CBB44A0E9C5E9289145},$$

and the independently recomputed digest is

$$d_0 = \texttt{417B0FA0B141CC4DD22537418F34B941FBC7DEE0}.$$

The resulting state update is

$$s_0 \oplus d_0[0 : 10] = \texttt{DDC04B00588425654360},$$

and the emitted bytes are

$$d_0[10 : 20] = \texttt{37418F34B941FBC7DEE0}.$$

Across the four iterations, the reconstructed 32-byte output is

$$\begin{aligned}\texttt{RAND\_bytes}[0 : 32] = {} & \texttt{37418F34B941FBC7DEE0E89B6BB9D586} \\ & \texttt{8E164287F33F1BF5DB9173304C0BC1AC}.\end{aligned}$$

This value matches the observed `stage="after"` output in the `ssleay_rand_bytes` trace. The replay therefore validates the post-stir byte-generation loop without treating the Windows `RAND_poll` inputs as a naive concatenated buffer.

12.8 Wallet-Level Check from the Observed `RAND_bytes` Output

The previous subsection validates the OpenSSL-side reconstruction of the 32-byte value returned by `ssleay_rand_bytes`. A final artifact checks how this observed value maps to Bitcoin wallet material.

The replayed OpenSSL output is:

$$\begin{aligned}r = {} & \texttt{37418F34B941FBC7DEE0E89B6BB9D586} \\ & \texttt{8E164287F33F1BF5DB9173304C0BC1AC}.\end{aligned}$$

This value matches the observed `stage="after"` output in the `ssleay_rand_bytes` trace. In the wallet proof, it is interpreted as a 32-byte secret candidate:

$$\texttt{secret} = r.$$

The verification script then computes the corresponding Wallet Import Format encodings using Bitcoin's Base58Check convention. For completeness, both the historical uncompressed form and the compressed-key convention are checked:

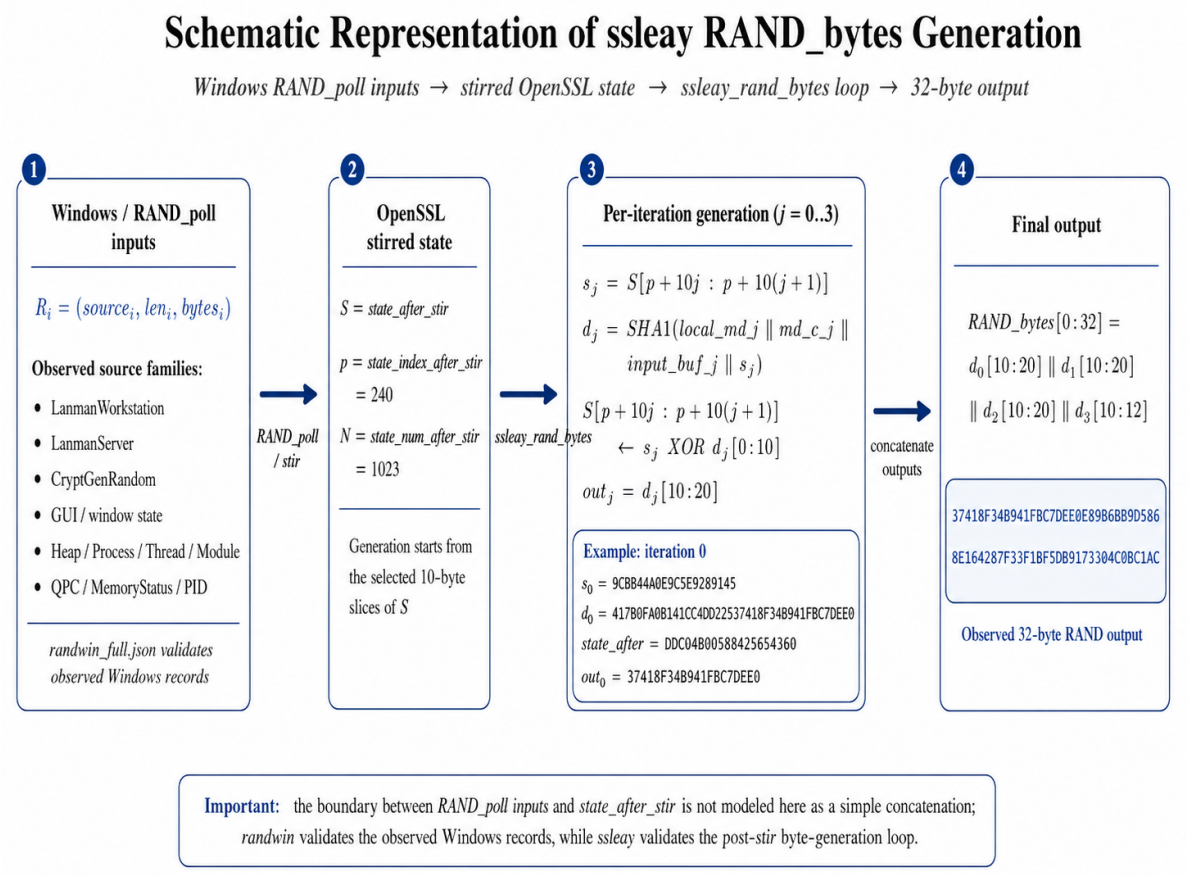

Figure 5 – Schematic representation of the experimentally validated boundary between Windows `RAND_poll` inputs, the stirred OpenSSL state, and the post-stir `ssleay_rand_bytes` generation loop. The Windows records are validated structurally by the `randwin` module, while the `ssleay` module independently recomputes the SHA-1 digests, state XOR updates, and final 32-byte output.

Encoding	Value
WIF, uncompressed	`5JEd2MfroiV4cMzmtn2KCQEDYCmF2q6vSBUKj216731475hZVdP`
WIF, compressed	`Ky57zzEWqC6rw5g19XioKKXLLivGw5K8r9DbeTRPXdqEhTEedzBf`

The same artifact also checks the corresponding P2PKH address derivations from the public keys recorded in the expected wallet-side artifact. The uncompressed public key path gives:

`13n3au9sg4pxqS2p7pweR1eGiGftEpRQyb`.

The compressed public key path gives:

`1LM9HmujdtL69m82oR1xUqTcK1xhpreHEA`.

Figure 6 summarizes this final wallet-level consistency check.

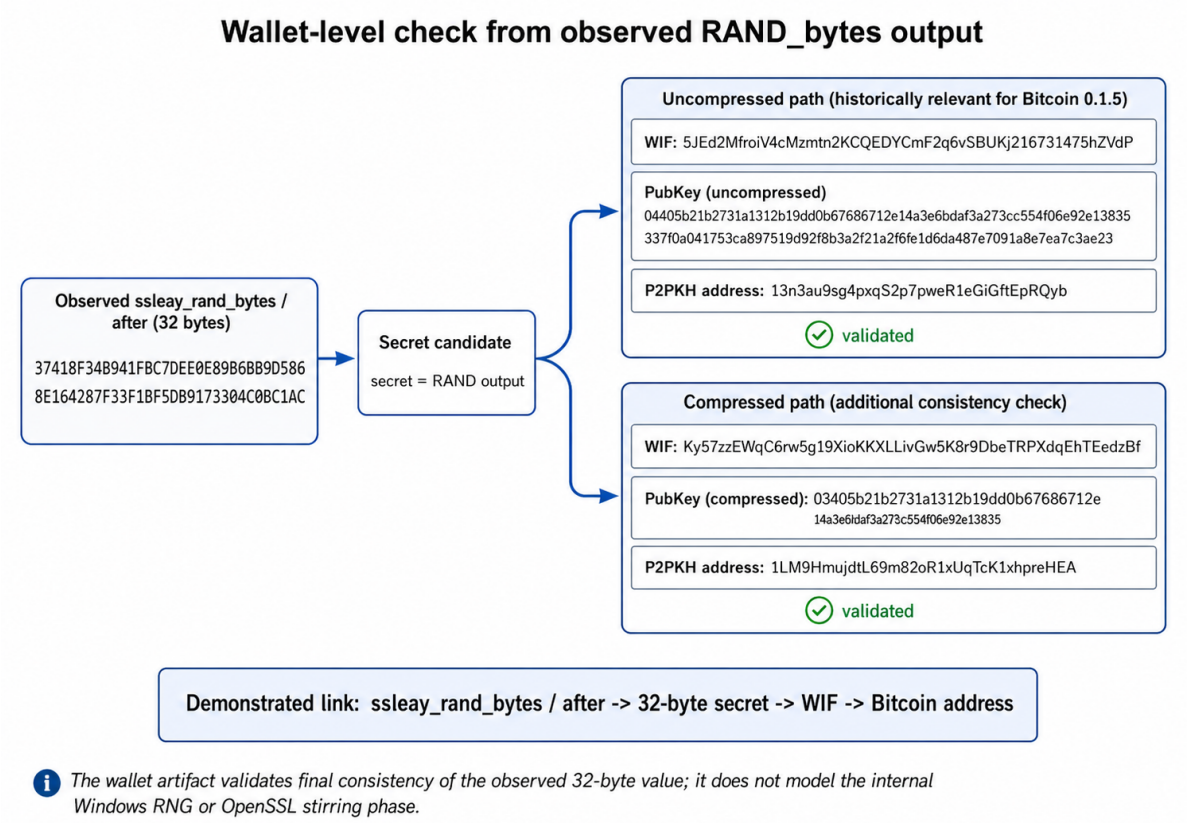

Figure 6 – Wallet-level consistency check from the observed `ssleay_rand_bytes/after` output. The observed 32-byte OpenSSL output is interpreted as a secret candidate, encoded as WIF, and checked against the corresponding P2PKH addresses. The uncompressed path is the historically relevant Bitcoin 0.1.5 convention; the compressed path is included only as an additional consistency check.

The reproduced command is:

```
python3 wallet_proof.py sample01/prng_log_excerpt.jsonl
```

The relevant validation result is:

```
RAND bytes from ssleay_rand_bytes/after (32):
 37 41 8F 34 B9 41 FB C7 DE E0 E8 9B 6B B9 D5 86
```

```
 8E 16 42 87 F3 3F 1B F5 DB 91 73 30 4C 0B C1 AC

SECRET candidate:
 37418F34B941FBC7DEE0E89B6BB9D5868E164287F33F1BF5DB9173304C0BC1AC

Match RAND==SECRET: YES
WIF uncompressed OK: YES
WIF compressed OK  : YES
Address uncomp OK  : YES
Address comp OK    : YES
```

This check should be interpreted narrowly. It validates the final wallet-side consistency of the observed 32-byte value: the value extracted from `ssleay_rand_bytes/after` is identical to the secret candidate used in the wallet proof, its WIF encodings match the expected values, and the recorded public keys reproduce the expected P2PKH addresses.

It does not add a new claim about the internal Windows RNG or about the OpenSSL stirring phase. Those parts are handled separately by the `randwin` and `ssleay` modules. The purpose of the wallet artifact is to close the final demonstrative link:

$$\texttt{ssleay\_rand\_bytes/after} \rightarrow \texttt{32-byte secret candidate} \rightarrow \texttt{WIF} \rightarrow \texttt{P2PKH address}.$$

For historical Bitcoin 0.1.5, the uncompressed public key path is the relevant wallet convention. The compressed path is included only as an additional consistency check for the same 32-byte scalar value.

12.9 Independent Check with the Historical Bitcoin Client

As an additional application-level consistency check, the `wallet.dat` produced during the experimental run was opened with an unmodified historical Bitcoin 0.1.5 client in the Windows XP test environment. The address displayed by the client matches the uncompressed P2PKH address derived offline from the observed `ssleay_rand_bytes/after` output.

This check is independent of the replay scripts used in the previous sections. The offline reconstruction yields a 32-byte secret candidate, its corresponding uncompressed WIF encoding, and the expected P2PKH address; the historical client then reads the wallet file and exposes the same address through its normal interface. This provides an application-level consistency check linking the reconstructed random output to the wallet artifact recognized by Bitcoin 0.1.5.

This observation should be interpreted narrowly. It does not provide a new claim about the internal Windows RNG or about OpenSSL's stirring mechanism. Those components are addressed separately by the `randwin` and `ssleay` artifacts. The present check only confirms that the reconstructed 32-byte value is consistent with the wallet material later recognized by the historical Bitcoin client.

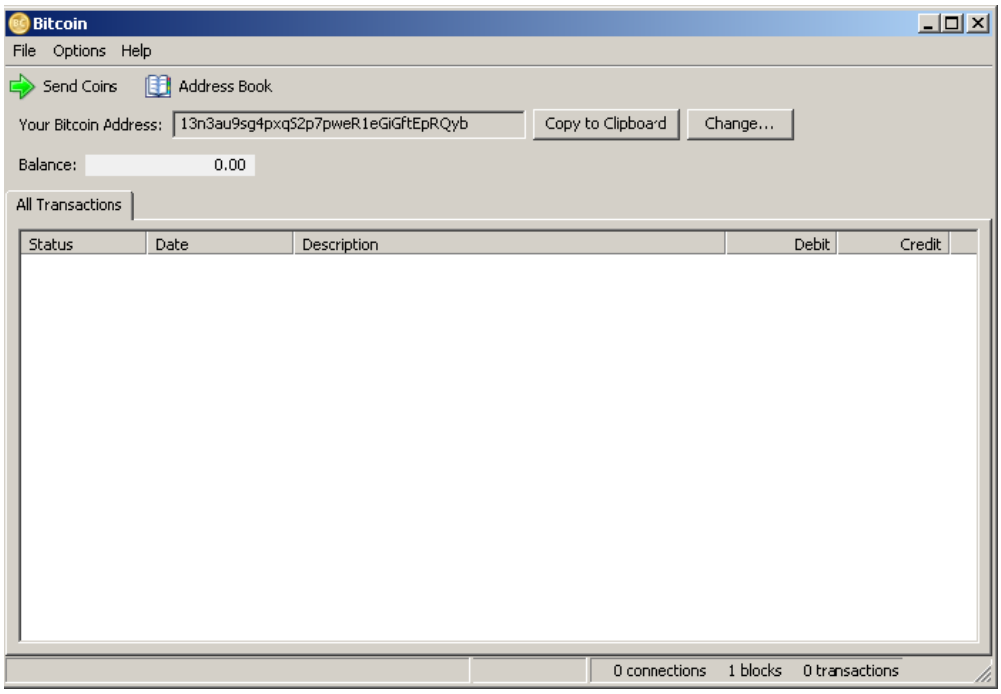

Figure 7 – Independent application-level consistency check. The historical Bitcoin 0.1.5 client, run in the Windows XP experimental environment, opens the `wallet.dat` produced during the experimental run and displays the same uncompressed P2PKH address as the one derived offline from the observed `ssleay_rand_bytes`/`after` output. This figure illustrates agreement at the application layer; it is not itself a reconstruction of the internal Windows RNG path or of the OpenSSL stirring phase.

12.10 Campaign Results

The captures retained for the main campaign contain between `441` and `731` entries per execution. These values correspond to the normal regime observed in the retained `50`-execution campaign.

For scalar or quasi-scalar sources, the analysis retains the modal support. In this support, these sources appear twice per run. No execution is excluded from the retained campaign.

During broader exploratory runs outside the retained campaign, an exceptional third occurrence was observed once for one of these scalar sources. This event coincided with severe debugger-induced slowdown and destabilized the laboratory setup. It is therefore treated as an instrumentation artifact and is not retained in the modal-support analysis.

Source	Modal occ.	Retained bytes	Variable bytes	Variable density
`CryptGenRandom`	2	128	128	1.000
`GetForegroundWindow`	2	8	6	0.750
`QueryPerformanceCounter`	2	16	12	0.750
`GetCurrentProcessId`	2	8	4	0.500
`GetQueueStatus`	2	8	2	0.250
`GlobalMemoryStatus`	2	64	6	0.094
`LanmanServer`	2	136	8	0.059
`LanmanWorkstation`	2	432	16	0.037
`GetCursorInfo`	2	0	0	0.000

Within this framework, `CryptGenRandom` appears twice per execution, as two `64`-byte blocks. All comparable positions in these blocks vary at least once across the retained executions.

The other scalar or quasi-scalar sources exhibit heterogeneous profiles. `GetForegroundWindow` and `QueryPerformanceCounter` show high relative variability, but over very small retained byte volumes. `GetCurrentProcessId` and `GetQueueStatus` also vary over small volumes.

By contrast, `GlobalMemoryStatus`, `LanmanServer`, and `LanmanWorkstation` contribute larger structured inputs but with lower byte-wise variation density. In particular, `LanmanWorkstation` retains `432` comparable bytes but only `16` variable byte positions in this campaign.

12.11 Visualization of Per-Source Contributions

The following figures summarize the collected volumes, variable volumes, and variation densities by source. They complement the previous table and make it possible to distinguish the raw collected volume from the variability actually observed.

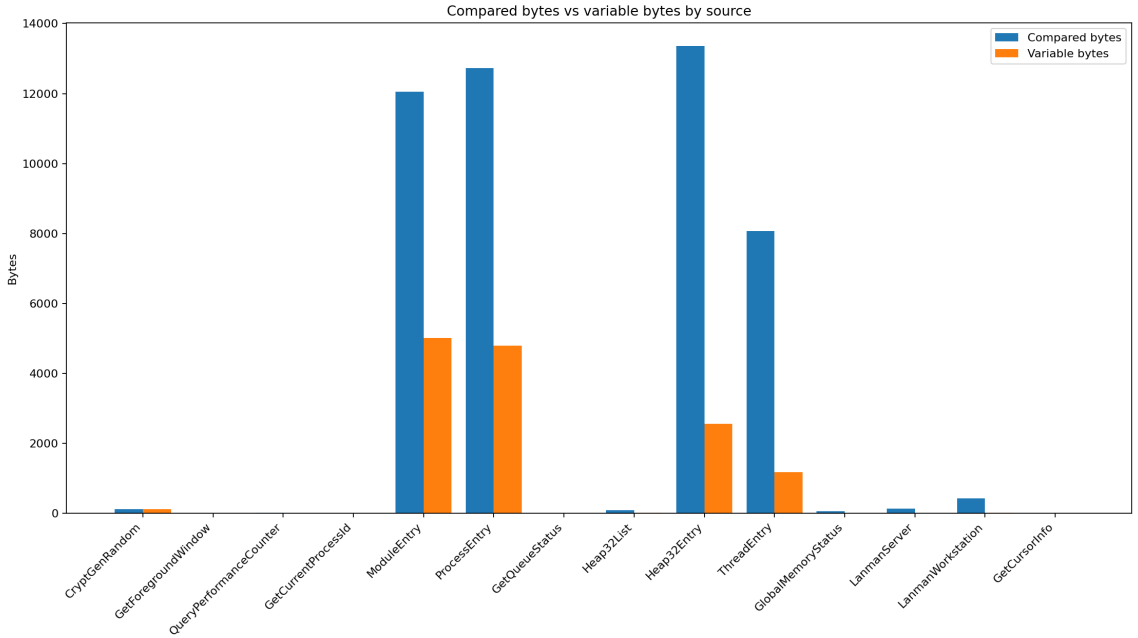

Figure 8 – Total volume and variable volume by source in the `RAND_poll` campaign.

Figure 8 shows that enumeration sources such as `ProcessEntry`, `ModuleEntry`, `Heap32Entry`, and `ThreadEntry` dominate in total volume. Their variable contribution exists, but remains structural and must be distinguished from a direct entropy estimate.

Figure 9 highlights that `CryptGenRandom` exhibits maximal variation density within the observed modal support. Sources such as `QueryPerformanceCounter` or `GetForegroundWindow` also exhibit high relative density, but over much smaller volumes.

Figure 10 shows that the dominant category is partial presence. This result mainly reflects enumeration sources, whose cardinality and order may vary from one execution to another. It confirms that the variability observed in `RAND_poll` combines content changes, structural effects, and presence effects.

12.12 Enumeration Sources

System enumeration sources exhibit substantial structural variability. Their cardinality varies across executions:

— `ProcessEntry`: 18 to 55 elements;
— `ModuleEntry`: 15 to 32 elements;
— `Heap32Entry`: 160 to 420 elements;

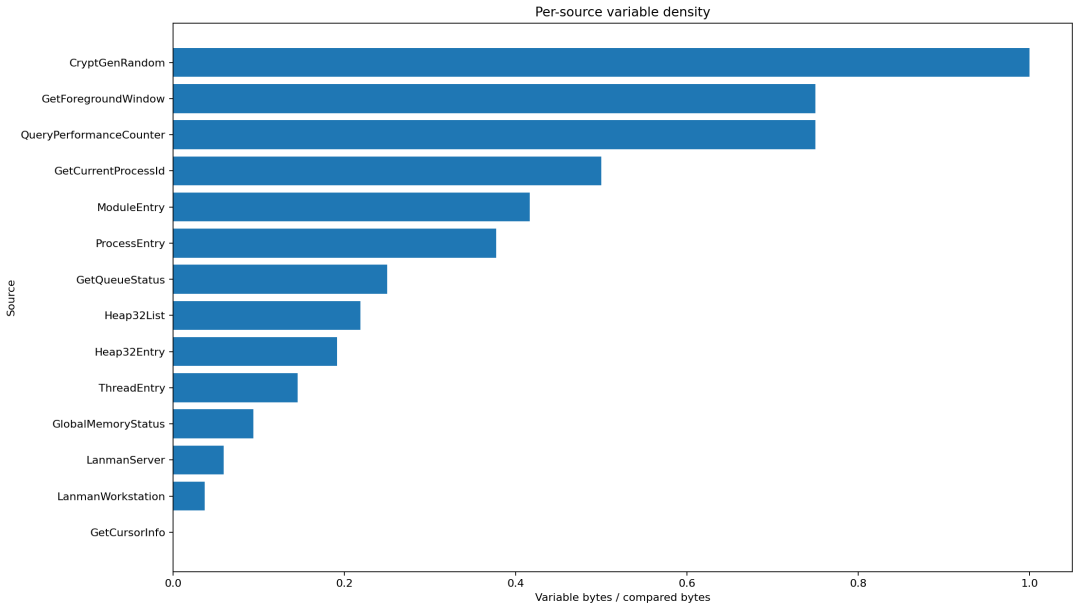

Figure 9 – Variation density by source, defined as the ratio between variable bytes and comparable bytes.

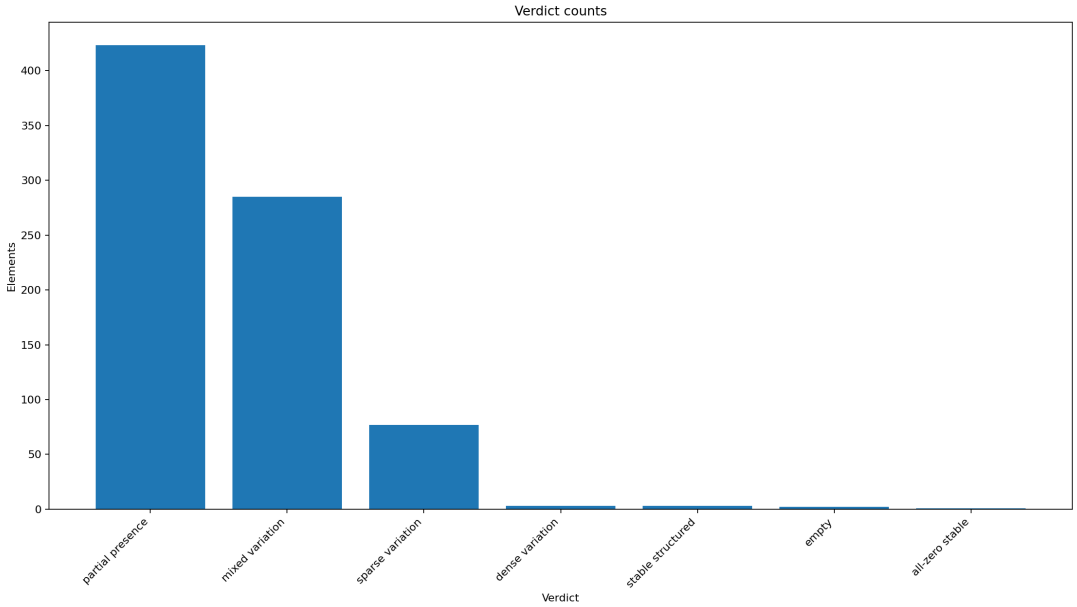

Figure 10 – Distribution of variation categories observed in aligned elements.

— `ThreadEntry`: 158 to 440 elements.

These sources contribute a large absolute volume of variable bytes. However, this variability does not necessarily correspond to independent variation in the data: it may reflect ordering effects, presence or absence of objects, or differences in internal structure.

12.13 Distinction from Kernel Collection

OpenSSL collection through `RAND_poll` must be distinguished from the internal mechanism triggered by `CryptGenRandom`. In the latter case, the traces show a sequence of `ZwQuerySystemInformation` calls:

```
5, 3, 7, 2, 0x21, 0x2d, 8, 0x17
```

This sequence is dynamically observed and corresponds to the kernel collection phase feeding the aggregated buffer used by `VeryLargeHashUpdate`.

12.14 Scope of the Observations

The measurements presented characterize the empirical variability of the sources collected in `RAND_poll`. They do not constitute a direct estimate of cryptographic entropy, nor a min-entropy measurement.

More precisely:

— observed variation of a byte indicates a change at least once across executions, without information about its distribution;

— dependencies between sources are not modeled;

— some contributions reflect system structure rather than unpredictable fluctuations.

Within this framework, the call to `CryptGenRandom` stands out empirically through dense variability across all observed bytes. This observation does not, however, prejudge the intrinsic cryptographic quality of the underlying generator, which belongs to the separate analysis of the internal Windows path.

13 Limitations and Residual Uncertainties

13.1 No Complete Formal Proof

This study does not claim to provide a complete formal proof of the Windows XP RNG. It provides a trace-based reconstruction, with bit-level replay of several sub-stages. The results should be read as a precise experimental mapping of the observed path, not as an exhaustive mathematical specification of all possible versions or configurations.

13.2 Nature of Offline Replay

Offline replay does not consist in mechanically replaying captured memory dumps. The dumps serve as experimental inputs and reference values, but the transformations are reimplemented at the algorithmic level in an independent environment.

Thus, for `VeryLargeHashUpdate`, the validator reproduces the underlying SHA-1 logic, including internal compression operations. Similarly, on the provider side, local validations concern the primitives themselves, for example SHA-1 compression rounds, MD4 operations, XOR operations, and RC4-like stages, including the complete KSA and stream generation when the required buffers are available.

This distinction is important: bit-level agreement does not result from a simple comparison between captures, but from an executable reconstruction of the observed transformations. The remaining limitations therefore concern the complete temporal articulation of certain persistent provider states, not the absence of an independent implementation of the primitives already closed.

Example (RC4 KSA). The following trace illustrates the offline reconstruction of the RC4 Key Scheduling Algorithm (KSA) from captured inputs.

```
=== INPUTS ===
K sha1     : 0f0fd7ef8f764c12ae929c63d0da2b2a13f5c054
S0 sha1    : 4916d6bdb7f78e6803698cab32d1586ea457dfc8
S_obs sha1 : 95e0416a8889440ba2bdceab8b018622b0f56c10

i=00 K=35 Si=00 j:00->35 Sj_before=35 swap(00,35) Si'=35 Sj'=00
i=01 K=72 Si=01 j:35->a8 Sj_before=a8 swap(01,a8) Si'=a8 Sj'=01
i=02 K=13 Si=02 j:a8->bd Sj_before=bd swap(02,bd) Si'=bd Sj'=02
i=03 K=65 Si=03 j:bd->25 Sj_before=25 swap(03,25) Si'=25 Sj'=03
i=04 K=a6 Si=04 j:25->cf Sj_before=cf swap(04,cf) Si'=cf Sj'=04
i=05 K=6f Si=05 j:cf->43 Sj_before=43 swap(05,43) Si'=43 Sj'=05
i=06 K=0f Si=06 j:43->58 Sj_before=58 swap(06,58) Si'=58 Sj'=06
i=07 K=25 Si=07 j:58->84 Sj_before=84 swap(07,84) Si'=84 Sj'=07
i=08 K=db Si=08 j:84->67 Sj_before=67 swap(08,67) Si'=67 Sj'=08
i=09 K=d4 Si=09 j:67->44 Sj_before=44 swap(09,44) Si'=44 Sj'=09
i=0a K=de Si=0a j:44->2c Sj_before=2c swap(0a,2c) Si'=2c Sj'=0a
i=0b K=94 Si=0b j:2c->cb Sj_before=cb swap(0b,cb) Si'=cb Sj'=0b
...
i=fa K=20 Si=19 j:03->3c Sj_before=ed swap(fa,3c) Si'=ed Sj'=19
i=fb K=e6 Si=c8 j:3c->ea Sj_before=f8 swap(fb,ea) Si'=f8 Sj'=c8
i=fc K=0a Si=8e j:ea->82 Sj_before=21 swap(fc,82) Si'=21 Sj'=8e
i=fd K=78 Si=fd j:82->f7 Sj_before=3b swap(fd,f7) Si'=3b Sj'=fd
i=fe K=c1 Si=fe j:f7->b6 Sj_before=0e swap(fe,b6) Si'=0e Sj'=fe
i=ff K=83 Si=ff j:b6->38 Sj_before=44 swap(ff,38) Si'=44 Sj'=ff

=== FINAL ===
match      : True
j_final    : 38
S_calc sha1: 95e0416a8889440ba2bdceab8b018622b0f56c10
S_obs  sha1: 95e0416a8889440ba2bdceab8b018622b0f56c10
```

```
=== BIT CHECK ===
BIT-EXACT MATCH (2048 bits)
```

This final check confirms that the offline KSA implementation reproduces the entire 256-byte RC4 state exactly, i.e. `2048` bits, rather than merely matching a few intermediate bytes.

13.3 QSI-to-Pool Attribution

The useful reconstruction of the pool for `VeryLargeHashUpdate` is complete in the validated campaigns: the `0x258` bytes of the captured pool are segmented, consumed by `VeryLargeHashUpdate`, and replayed offline with bit-exact agreement for `seedbase_after`.

The residual limitation therefore no longer concerns the pool $\rightarrow$ `VeryLargeHashUpdate` replay, but the fine-grained semantic attribution of some pool bytes to their exact origin. Classes 3, 7, 2, `0x21`, and `0x2d` appear as contiguous and directly identifiable copies. Other regions correspond to fragments, zeros, metadata, or residues not yet certainly linked to a single system source.

This distinction is important: the exact origin of some bytes remains partially documented, but their effective value in the pool is captured and integrated into the full `VeryLargeHashUpdate` replay.

13.4 Function G

The upstream construction of `state20` remains the main lock. Recent campaigns show the final projection:

$$\texttt{provider\_source\_structure} \rightarrow \texttt{local region} \rightarrow \texttt{provider\_state20\_slot} \rightarrow \texttt{state20}.$$

This observation strongly reduces the scope of G, because `state20` is not produced by the final block. It is prepared upstream and then copied into the global slot consumed by H.

The remaining uncertainty concerns the initialization and update of the persistent provider structure used as the source. This part is not yet replayed bit-for-bit from `seedbase_after`.

13.5 Temporal Composition of the Provider

The MD4, SHA-1, RC4, XOR, KSA/PRGA primitives and the 256-byte circular buffer are identified and locally validated at several points. The link between the buffer returned by IOCTL `0x390008` and the key used by the RC4 KSA is also observed.

However, the complete temporal composition of these primitives, notably across several invocations and successive provider-state updates, remains partially open. This is what still prevents a complete replay of:

$$\texttt{seedbase\_after} \rightarrow \texttt{provider\_state} \rightarrow \texttt{state20}.$$

By contrast, the following sub-stages are closed in the validated runs:

$$\texttt{pool} \rightarrow \texttt{VeryLargeHashUpdate} \rightarrow \texttt{seedbase\_after},$$

$$(\texttt{state20}, \texttt{aux20}) \rightarrow H \rightarrow \texttt{out40},$$

$$\texttt{out40}[0 : 0x20] \rightarrow \texttt{CryptGenRandom}_{out}.$$

13.6 Virtualization

The experimental environment relies on a Windows XP SP3 virtual machine. Virtualization may alter some hardware-dependent sources, such as timings, counters, interrupts, or device characteristics.

This limits quantitative conclusions about the actual entropy available in a historical physical environment. It does not affect the structural analysis of the observed transformations, nor the bit-level equalities validated in the studied environment.

13.7 Scope of Observed Anchors

The anchors mentioned in this study correspond to the binaries, modules, and load configurations observed in the campaigns. They serve as experimental references for the traces, but should not be interpreted as stable or universal interfaces.

Another Windows version, another service pack, another provider, or a different memory load configuration may move these anchors or modify some internal paths.

13.8 No Exploitation Claim

This work does not demonstrate a practical attack against historical keys. The connection with OpenSSL and Bitcoin concerns the execution chain and historical relevance of the path:

$$\texttt{Bitcoin} \rightarrow \texttt{OpenSSL} \rightarrow \texttt{CryptGenRandom} \rightarrow \texttt{Windows XP RNG}.$$

It does not constitute proof of Bitcoin key compromise, nor a direct estimate of the effective entropy of the OpenSSL state.

14 Conclusion

This study describes `CryptGenRandom` on Windows XP SP3 as a hybrid multi-stage architecture whose effective behavior can be reconstructed through dynamic instrumentation and offline replay. The observed pipeline combines user-space provider logic, kernel collection, aggregation into a `0x258`-byte pool, SHA-1-based mixing through `VeryLargeHashUpdate`, provider state updates, MD4/SHA-1/RC4 primitives, circular-buffer operations, a final block based on SHA-1 compression, and a final copy to the user output buffer.

A first central result is the complete validation of the kernel-side `VeryLargeHashUpdate` transformation from captured pools. The `0x258`-byte pool, its segmentation into four `0x96`-byte blocks, and the update of `seedbase` are reconstructed and replayed offline. In the validated campaigns, the transformation

$$\texttt{pool} \rightarrow \texttt{VeryLargeHashUpdate} \rightarrow \texttt{seedbase\_after}$$

is closed bit-for-bit. The fine-grained semantic attribution of some pool bytes remains partially open, in particular for fragmentary or structurally variable sources. This limitation affects the interpretation of certain pool regions, but not the validity of the replay from the captured pool to `seedbase_after`.

A second result is the closure of the provider's final generation phase. The local buffer observed after the provider-local transformation is reused as `aux20` at the entry of the final block. The final block $H(\texttt{state20}, \texttt{aux20})$ is reproduced bit-for-bit, and the first `0x20` bytes of `out40` are copied to the `CryptGenRandom` output. This establishes a precise replay boundary: once `state20` and `aux20` are known, the final `CryptGenRandom` output is fully reproducible.

A third result is the substantial reduction of the remaining provider-state problem, denoted G. The traces show that `state20` is prepared upstream and copied into the provider global slot before entry into the final block, from a persistent provider structure. The residual uncertainty is therefore not diffuse across the whole pipeline. It is localized to the initialization and update of the provider persistent state, and more specifically to the mapping

$$\texttt{seedbase\_after}, \texttt{provider state} \rightarrow \texttt{state20}.$$

This is the principal component that remains open for a fully offline replay from `seedbase_after` to the final `CryptGenRandom` output.

Beyond these closures, the study shows that the internal transformations are not merely observed as memory transitions, but reimplemented. Primitives such as SHA-1 compression, MD4, and RC4, including the RC4 key-scheduling algorithm, are reproduced offline from captured inputs, with bit-level agreement for internal states. This distinguishes the reconstruction from a simple dump-to-dump comparison: the replay validates an effective algorithmic model of the observed pipeline.

The OpenSSL and Bitcoin artifacts provide a historical and application-level context for this reconstruction. On Windows, OpenSSL 0.9.8h can incorporate `CryptGenRandom` output into its internal RAND state through the Windows `RAND_poll/RAND_screen` path, together with other user-space system sources. The `randwin` traces document these Windows-side records structurally, while the `ssleay` replay validates the post-stir `ssleay_rand_bytes` generation loop from the stirred OpenSSL state to the observed 32-byte output. The wallet-level artifact then checks that this observed output is consistent with the corresponding WIF encoding and P2PKH address. Finally, opening the resulting `wallet.dat` with the historical Bitcoin 0.1.5 client provides an independent application-level consistency check: the unmodified client displays the same uncompressed address as the one derived offline.

These observations should be interpreted narrowly. The study does not claim a practical attack, key compromise, or general break of Bitcoin wallets. Nor does it claim a complete entropy assessment of all inputs collected by OpenSSL or Windows. The measurements characterize

observed variability, trace-level structure, and replayability of specific components in a controlled Windows XP SP3 environment. In particular, byte variation across runs must not be confused with a min-entropy estimate.

The main contribution of this work is therefore methodological and experimental. It transforms a complex historical RNG implementation into an instrumented pipeline, closes major segments by bit-exact replay, and circumscribes the remaining uncertainty to a specific provider-state transition. This provides a reproducible basis for analyzing legacy RNG architectures and highlights the importance of studying effective implementations beyond high-level descriptions, static reconstructions, or standardized abstractions.

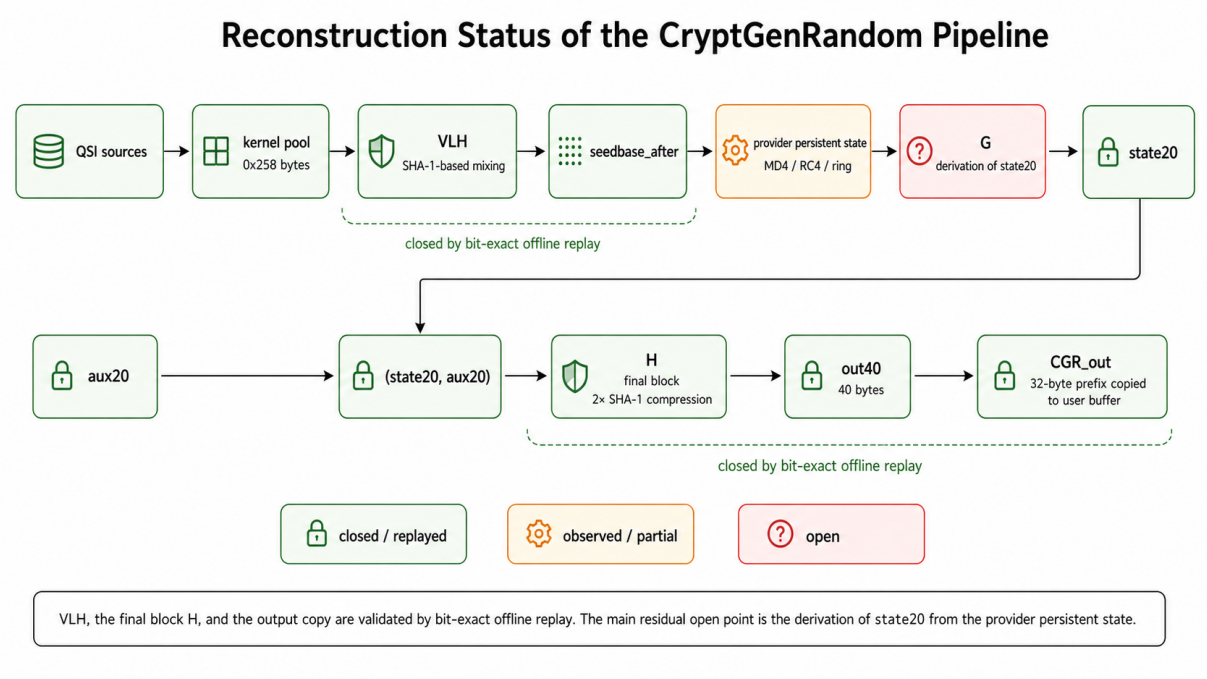

Figure 11 – Reconstruction status of the observed `CryptGenRandom` pipeline. The kernel pool, `VeryLargeHashUpdate`, the final provider block H, and the output copy are validated by bit-exact offline replay. The main residual open point is the derivation of `state20` from the provider persistent state.

Acknowledgments. The author thanks Benny Pinkas for helpful discussions and for comments that helped clarify the positioning of this work with respect to prior analyses of random number generators. The author also thanks K. Paterson for pointing out relevant prior work, including [7], and for discussions that helped refine the historical and empirical framing of the study.

The author acknowledges helpful exchanges with Niels Ferguson regarding Windows RNG implementations and practical aspects of deployed random number generators.

These exchanges contributed to improving the presentation and contextualization of the results, but do not constitute validation, endorsement, peer review, or co-authorship.

The author also thanks his family and friends for their support throughout this work. In particular, the author dedicates a special thought to Peter Stiehl, white-hat hacker and friend, who passed away on April 28, 2019, at the age of 35, from a rare cancer. Among many things, he taught the author a lesson that remained central to this work: always read the manual.

All experiments, instrumentation, reverse-engineering work, offline replay implementations, analyses, and possible errors remain solely the responsibility of the author.

References

[1] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of rc4 in tls. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, 2013.

[2] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. On the practical exploitability of dual ec in tls implementations. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, 2014.

[3] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the random number generator of the windows operating system. IACR Cryptology ePrint Archive, Report 2007/419, 2007.

[4] Niels Ferguson. Extensions of single-term coins. In *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 292–301. Springer, 1994.

[5] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, New York, 2003.

[6] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE, 2006.

[7] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*. USENIX Association, 2012.

[8] National Institute of Standards and Technology. Recommendation for random number generation using deterministic random bit generators. Special Publication 800-90A Revision 1, NIST, 2015.

[9] National Institute of Standards and Technology. Recommendation for the entropy sources used for random bit generation. Special Publication 800-90B, NIST, 2018.

[10] National Institute of Standards and Technology. Recommendation for random bit generator constructions. Special Publication 800-90C, NIST, 2024.

[11] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, second edition, 1996.

[12] Dan Shumow and Niels Ferguson. On the possibility of a back door in the nist sp800-90 dual ec prng. CRYPTO 2007 Rump Session, 2007.

A Summary Status of Sub-Problems

Sub-problem	Status	Comment
`ZwQuerySystemInfo` sequence	Experimentally closed	The sequence observed in the long paths is stable: $5,\ 3,\ 7,\ 2,\ 0x21,\ 0x2d,\ 8,\ 0x17.$ It is observed dynamically in the context of the runs studied.
`ZwQuerySystemInfo` → pool `0x258`	Partially attributed	The complete pool is captured. Classes 3, 7, 2, `0x21`, and `0x2d` appear as contiguous copies. Other bytes correspond to fragments, zeros, metadata, or residual values whose fine-grained semantic origin remains partially documented.
Captured pool → `VeryLargeHashUpdate`	Closed	The `0x258` bytes of the captured pool are segmented into four `0x96`-byte blocks and consumed by `VeryLargeHashUpdate`. Offline replay reproduces the observed values.
`VeryLargeHashUpdate` → `seedbase_after`	Closed	The values `seed0'` through `seed3'` and `seedbase_after` match the dumps observed in the validated campaigns.
Kernel-provider bridge → RC4 key	Locally closed	The `0x100`-byte buffer returned by the kernel-provider bridge is observed as the key at the RC4 KSA entry in the same run.
RC4 KSA	Locally closed	KSA replay from the captured key reproduces the RC4 state observed after initialization in the locally validated runs.
Provider MD4 → state injection	Locally closed	The MD4 digest computed offline matches the observed digest and its XOR injection into the provider state for the captured buffers.
Provider local path → `aux20`	Experimentally closed	`AFTER_LOCAL` is passed unchanged to the entry of the final provider block.
Immediate projection to `state20`	Closed	A `20`-byte copy into the provider state slot is observed as a copy of five `32`-bit words.
Provider source structure → source20	Reduced but open	The immediate source of `state20` originates from a provider global source structure. The exact producer and upstream update of this structure remain to be reconstructed.
`seedbase_after` → `state20`	Open	This is the main remaining lock. The local primitives have been identified, but their full temporal composition has not yet been replayed bit-for-bit.
(`state20`, `aux20`) → `out40`	Closed	The final provider block is replayed offline with exact reproduction of the observed `40` bytes.

Sub-problem	Status	Comment
$\texttt{out40}[0 : 0x20] \rightarrow \texttt{CryptGenRandom}_{out}$	Closed	The final copy to the user buffer is observed and validated by direct comparison.

B Hashes of Analyzed Artifacts

The following hashes identify the binaries used in the experimental environment (Windows XP SP3, Bitcoin 0.1.5, and OpenSSL 0.9.8h).

They are provided for reproducibility and traceability. MD5 is used here as a file identifier, not as a cryptographic security mechanism.

File	MD5
`advapi32.dll`	`bab489a5fe26f2d0c910cf7af7e4cf92`
`kernel32.dll`	`c24b983d211c34da8fcc1ac38477971d`
`ksecdd.sys`	`1705745d900dabf2d89f90ebaddc7517`
`rsaenh.dll`	`54dae3ea34802b4ed9ae1c6b1209fa56`
`libeay32.dll`	`192ef960bd269b499097fb154ef2b6f8`
`mingwm10.dll`	`d6109c0e39f2d4d2fb86505159021b3f`
`bitcoin.exe`	`307ad86c412c02abeb821afbb900355b`
`test_rng.exe`	`0074255b6fc31938031d39f977182f49`

These hashes correspond to the binaries actually loaded in the experimental environment. A version change (service pack, patch level, cryptographic provider, or system configuration) may modify the observed addresses, offsets, and internal paths.

Results that depend on absolute addresses should therefore be interpreted relative to these specific artifacts.

`test_rng.exe` is an experimental tool used to trigger and observe calls to `CryptGenRandom` in a controlled environment. Its behavior was compared with the behavior observed during execution of `bitcoin.exe`, in particular with respect to call sequence, buffer sizes, and the provider-side execution path.

The observations show correspondence on these aspects in the experimental environment studied. This validation does not constitute proof of complete execution identity between the two programs, but it establishes that `test_rng.exe` reproduces the path relevant to the analysis of `CryptGenRandom`.

C Campaign Summary

C.1 QSI / Pool Campaigns

The QSI campaigns capture the outputs of the system-information classes invoked in the long path and compare them with the `0x258`-byte pool transmitted to `VeryLargeHashUpdate`.

They establish three distinct facts:

— the `ZwQuerySystemInformation` class sequence is stable in the observed runs;
— the complete `0x258`-byte pool is captured before `VeryLargeHashUpdate`;
— some pool regions are directly attributable to `ZwQuerySystemInformation` outputs, whereas others correspond to fragments, zeros, metadata, or residual values whose semantic origin is not yet fully assigned.

The residual limitation concerns the fine-grained origin of certain bytes, not their effective value in the `VeryLargeHashUpdate` replay.

C.2 VLH Campaigns

The `VeryLargeHashUpdate` campaigns capture:

— the complete pool;
— the segments S_0, S_1, S_2, and S_3;
— `seedbase_before`;
— `seedbase_after`.

Offline replay reproduces the four prime seed values and the final concatenation:

$$\texttt{seedbase\_after} = \texttt{seed0'} \parallel \texttt{seed1'} \parallel \texttt{seed2'} \parallel \texttt{seed3'}.$$

In the validated campaigns:

$$\texttt{vlh\_replay(seedbase\_before, pool)} = \texttt{seedbase\_after}.$$

C.3 Provider / P6 Campaigns

The provider campaigns capture the events `STATE20_WRITE`, `LOCAL_ENTER`, `AFTER_LOCAL`, and `FIPS_ENTRY`. They make it possible:

— to pair `aux20` with the finalized local buffer;
— to show that `state20` is copied to `68031958` before entry to the final block;
— to reduce the remaining lock G to the upstream construction of the provider source structure.

C.4 IOCTL / RC4 Campaigns

The IOCTL / RC4 campaigns capture the return of IOCTL `0x390008` and the entry to the RC4 KSA.

They locally validate:

$$\text{OutputBuffer}_{IOCTL} = K_{RC4}.$$

Then:

$$\text{KSA}(K_{RC4}) = S_{after}.$$

This validation establishes the role of the IOCTL buffer in provider-side RC4 initialization, but does not by itself close the complete relation from `seedbase_after`.

C.5 H Campaigns

The H campaigns capture:

— `state20`;
— `aux20`;
— the 64-byte blocks constructed before SHA-1 compression;
— `out40`;
— the source and destination of the final copy.

They validate:

```
out40_calc == out40_after
out40_after[:0x20] == final_dst32_after
```

The final block and the user-space copy are therefore closed bit-for-bit in the validated runs.

C.6 `RAND_poll`, QSI, and Variation Density Campaigns

To distinguish the mere presence of a source from its empirically observable contribution, two complementary campaigns were conducted. The first concerns OpenSSL `RAND_poll` collection under Windows over `50` independent executions. The second concerns the kernel collection feeding `CryptGenRandom`, using `5` QSI/VLH campaigns containing the files `qsi_class_*.bin`, `pool.bin`, `s0..s3.bin`, and `seedbase_before/after.bin`.

The goal of these campaigns is not to estimate cryptographic min-entropy directly, but to measure the positional variability observed across executions. The metrics are therefore descriptive: a byte or bit position is considered variable if it changes at least once among the aligned captures. This measure identifies fixed, quasi-fixed, structural, or highly mobile regions without assuming statistical independence between fields.

C.6.1 Validation Layers for the OpenSSL and Bitcoin Checks

The OpenSSL and Bitcoin checks are separated into distinct validation layers. The `randwin` artifacts validate the structure and variability of Windows-side records collected during `RAND_poll`. These records include raw byte blobs associated with Windows collection sources; they are not all interpreted as decoded Win32 structures.

The `ssleay` replay validates the post-stir byte-generation loop of `ssleay_rand_bytes`, after OpenSSL's internal state has already been stirred. It is therefore not a reconstruction of the complete OpenSSL stirring process from Windows inputs.

The wallet-level check starts only from an observed `RAND_bytes` output and verifies its consistency with the corresponding Bitcoin secret key, WIF encoding, public key, and P2PKH address. This check is a final consistency check on the observed output; it is not evidence about the internal Windows RNG or about the OpenSSL `RAND_poll` stirring phase.

Finally, the historical Bitcoin client check is an application-level consistency check showing that the resulting wallet artifact is accepted by an unmodified client. It does not constitute an additional reconstruction of the internal random-generation path.

Together, these checks relate the Windows RNG, OpenSSL, and Bitcoin layers experimentally, but they do not close the provider-side transition from `seedbase_after` to `state20`, nor do they imply a complete offline replay from Windows kernel collection to Bitcoin wallet material.

C.6.2 `RAND_poll` Campaign over 50 Executions

The `50` `RAND_poll` captures were reconstructed from the `randwin_data.bin` files and converted into `randwin_full.json` representations. The comparison was performed using file-level alignment, relying on the reconstructed logical file names associated with each collected element.

The raw captures contain between 441 and 731 entries per execution. No execution was excluded from this analysis. The observed entry-count range is treated as part of the measured behavior of the instrumented RAND_poll campaign.

The resulting comparison defines 794 logical elements. In total, the global comparison covers 47 100 aligned bytes, of which 13 753 vary across captures, corresponding to an overall byte-wise variation density of 29.20%.

Source	Elements	Compared bytes	Variable bytes	Variable density	Nonzero density
CryptGenRandom	2	128	128	1.000	1.000
GetForegroundWindow	2	8	6	0.750	0.750
QueryPerformanceCounter	2	16	12	0.750	0.750
GetCurrentProcessId	2	8	4	0.500	0.500
ModuleEntry	29	12 056	5 019	0.416	0.574
ProcessEntry	50	12 728	4 799	0.377	0.599
GetQueueStatus	2	8	2	0.250	0.250
Heap32List	6	96	21	0.219	0.365
Heap32Entry	371	13 356	2 557	0.191	0.316
ThreadEntry	320	8 064	1 175	0.146	0.201
GlobalMemoryStatus	2	64	6	0.094	0.719
LanmanServer	2	136	8	0.059	0.074
LanmanWorkstation	2	432	16	0.037	0.042
GetCursorInfo	2	0	0	0.000	0.000

The nonzero density column is not an inter-run variability metric. It reports the fraction of nonzero bytes observed inside the compared structures. A source may therefore have nonzero density while showing low or no byte-wise variation across executions.

The distribution of variability is highly heterogeneous. At source level, CryptGenRandom is the only source with fully dense byte-wise variation in this campaign: all 128 comparable byte positions vary at least once across the 50 executions. This is consistent with its role as the operating-system random output supplied to OpenSSL.

QueryPerformanceCounter, GetCurrentProcessId, GetForegroundWindow, and GetQueueStatus also show non-negligible relative variability, but over very small byte volumes. Their contribution should therefore be interpreted as compact run-dependent state, rather than as a large byte-volume contribution.

The enumeration-based sources account for most of the variable bytes in absolute terms. ModuleEntry, ProcessEntry, Heap32Entry, and ThreadEntry contain substantial variation, but this variation combines several effects: actual field changes, object presence or absence, enumeration-rank shifts, process and module identifiers, pointers, counters, strings, and fixed structure fields.

Finally, although LanmanWorkstation and LanmanServer are collected, they exhibit low inter-execution variability in this campaign. LanmanWorkstation contains 16 variable bytes out of 432, while LanmanServer contains 8 variable bytes out of 136.

This indicates that, in the observed environment, these sources behave as mostly quasi-static structured inputs across executions. Their inclusion in RAND_poll therefore produces only limited measurable byte-wise variability at the scale of this experiment.

However, this observation should not be interpreted as a definitive absence of contribution to the internal entropy pool, but rather as a limitation of the present measurement, which captures only inter-run variability at the byte level.

C.6.3 Enumeration Sources and Partial Presence

The 50-execution campaign contains a large number of partially present logical elements. This is especially visible for system enumeration sources, whose cardinality changes between executions. Consequently, an aligned element such as ProcessEntry#i, ModuleEntry#i, Heap32Entry#i, or

ThreadEntry#i should not automatically be interpreted as denoting the same semantic object in every run.

The element-level verdicts are as follows:

Verdict	Number of elements
partial presence	423
mixed variation	285
sparse variation	77
dense variation	3
stable structured	3
empty	2
all-zero stable	1

Partial presence is itself informative: it shows that RAND_poll captures not only scalar values, but also the instantaneous shape of the software environment. Methodologically, however, it limits direct entropy interpretation. A byte-wise difference at a fixed enumeration rank may reflect a genuine field change, an ordering effect, or the appearance or disappearance of an object.

For this reason, the byte-wise variation densities reported here should be read as empirical variability measurements, not as independent entropy estimates.

C.6.4 QSI/VLH Campaign over 5 Executions

The second campaign directly analyzes the kernel-side collection path feeding CryptGenRandom. Although several dozen QSI/VLH campaigns were conducted during the broader study, the quantitative comparison reported here is based on a fresh set of 5 executions, captured under a controlled and homogeneous setup. This conservative subset was selected while the scripting infrastructure for large-scale, automatically reproducible campaigns was still being finalized. The same methodology is intended to scale to hundreds of executions once this automation is complete.

This campaign compares the QSI classes observed in the ZwQuerySystemInformation path, as well as the 0x258-byte pool.bin buffer injected into VeryLargeHashUpdate. The observed QSI sequence is:

```
05, 03, 07, 02, 21, 2d, 08, 17
```

Across the 5 executions, the raw QSI files alone represent 80 000 aligned bit positions, of which 4 586 vary at least once, corresponding to a global density of 5.73%. When the derived files pool.bin, s0..s3.bin, and seedbase_before/after.bin are included, 90 880 bit positions are compared, of which 8 004 vary, corresponding to a global density of 8.81%.

File	Interpretation	Variable bits	Compared bits	Ratio
pool.bin	actual VLH input	1 108	4 800	0.231
seedbase_before.bin	state before VLH	602	640	0.941
seedbase_after.bin	state after VLH	600	640	0.938
s0.bin	pool segment	401	1 200	0.334
s1.bin	pool segment	281	1 200	0.234
s2.bin	pool segment	141	1 200	0.118
s3.bin	pool segment	285	1 200	0.238

The central result is pool.bin. The 600-byte pool contains 1 108 variable bit positions out of 4 800, i.e., 23.08%. After VLH conditioning, seedbase_after contains 600 variable bit positions out of 640. This near saturation must not be interpreted as entropy creation: it reflects the

avalanche effect of SHA-1 conditioning applied to a variable accumulated state and to structured QSI inputs.

The fact that `seedbase_before.bin` is itself variable at `94.06%` is particularly important. It shows that the kernel state used by `CryptGenRandom` is not a simple function of the instantaneous QSI values from the current execution. It is an accumulated state, dependent on prior system history and earlier calls.

Accordingly, the present `5`-execution campaign should be read as a controlled validation sample for the QSI/VLH reconstruction, not as a final statistical characterization of the kernel entropy path. Larger automated campaigns are required before making stronger quantitative claims about the distribution of variability across executions.

C.6.5 QSI Classes and Contribution to the Pool

The QSI classes do not contribute equally to the pool. Some portions are mapped exactly, whereas other contributions remain fragmentary in this campaign.

Class	Variable bits	Compared bits	Ratio	Mapping
03	81	384	0.211	exact
02	496	2 496	0.199	exact
21	7	128	0.055	exact
07	0	192	0.000	exact
2d	0	256	0.000	exact
05	1 416	28 224	0.050	frag.
08	1 394	24 256	0.057	frag.
17	1 192	24 064	0.050	frag.

Class	Name	Pool mapping
03	SystemTimeOfDayInformation	[0x50:0x80)
02	SystemPerformanceInformation	[0xa8:0x1e0)
21	SystemExceptionInformation	[0x1e8:0x1f8)
07	SystemDeviceInformation	[0x88:0xa0)
2d	SystemLookasideInformation	[0x200:0x220)
05	SystemProcessInformation	fragmentary
08	SystemProcessorPerformanceInformation	fragmentary
17	SystemInterruptInformation	fragmentary

The exactly mapped classes explain `584` variable bits of the pool:

```
81 + 496 + 7 + 0 + 0 = 584
```

The complete pool contains `1 108` variable bits. Therefore, `524` variable bits remain located in regions not yet precisely attributed to QSI classes, but compatible with fragmentary contributions from classes `05`, `08`, and `17`. This attribution remains to be confirmed by finer mapping; these `524` positions are therefore not counted as a certain contribution of any specific class.

C.6.6 Localization of Variability in the Pool

Range-level analysis shows that pool variability is not uniform. It is concentrated in a small number of regions.

Pool zone	Range	Variable bits	Ratio
unmapped prefix	0x000:0x050	320 / 640	0.500
SystemTimeOfDayInformation	0x050:0x080	81 / 384	0.211
gap	0x080:0x088	0 / 64	0.000
SystemDeviceInformation	0x088:0x0a0	0 / 192	0.000
gap	0x0a0:0x0a8	0 / 64	0.000
SystemPerformanceInformation	0x0a8:0x1e0	496 / 2 496	0.199
gap	0x1e0:0x1e8	17 / 64	0.266
SystemExceptionInformation	0x1e8:0x1f8	7 / 128	0.055
gap	0x1f8:0x200	10 / 64	0.156
SystemLookasideInformation	0x200:0x220	0 / 256	0.000
unmapped tail	0x220:0x258	177 / 448	0.395

Four regions concentrate almost all observed pool variability:

Region	Variable bits	Share of total pool variability
pool[0x000:0x050)	320	28.9%
pool[0x050:0x080)	81	7.3%
pool[0x0a8:0x1e0)	496	44.8%
pool[0x220:0x258)	177	16.0%
Total	1 074	96.9%

Thus, the direct QSI-to-pool mapping accounts for approximately 72% of the 0x258-byte pool, as measured empirically through exact byte-level reconstruction. This establishes a large deterministic backbone in the pool construction.

By refining the analysis and attributing additional variable regions (including the unmapped prefix and tail, as well as partially characterized segments), up to 96.9% of the observed variable bits in the pool can be localized to a small set of structured regions. This indicates that the variability of the pool is highly concentrated rather than uniformly distributed.

However, this extended attribution remains partly interpretative: some regions of the pool are not yet fully mapped to precise QSI fields or kernel structures. The QSI-to-pool reconstruction is therefore substantially advanced but not fully closed, and a residual portion of the pool remains to be formally characterized.

C.7 Software Reconstruction Environment

To document reproducibility of the experimental environment, the components used to reconstruct the Bitcoin/OpenSSL/MinGW environment were archived with their MD5 hashes. These hashes are not used as cryptographic security mechanisms; they serve here as practical identifiers to verify that the software artifacts manipulated during the campaigns correspond to the archived files.

Component	MD5
binutils-2.19.1-mingw32-bin.tar.gz	6bba3bd1bf510d152a42d0beeeefa14d
gcc-core-3.4.5-20051220-1.tar.gz	3be0d55e058699b615fa1d7389a8ce41
gcc-g++-3.4.5-20051220-1.tar.gz	99059fbaa93fa1a29f5571967901e11f
mingwrt-3.15.2-mingw32-dev.tar.gz	f24d63744af66b54547223bd5476b8f0
mingwrt-3.15.2-mingw32-dll.tar.gz	688866a2de8d17adb50c54a2a1edbab4
w32api-3.13-mingw32-dev.tar.gz	a50fff6bc1e1542451722e2650cb53b4
mingw32-make-3.81-20080326-2.tar.gz	8692c3c6967f7530a2ad562fe69781d2

Component	MD5
wxWidgets-2.8.11.zip	33eda5d65838279f4dfbb369b7c75fbd
strawberry-perl-5.8.8.2.zip	79b4148f26fb3a7e7c30c8956b193880
openssl-OpenSSL_0_9_8h.zip	368d680fe87f395f9d161a45d6248f4d
msysDTK-1.0.1.exe	f7aeebb16dc3b0f19b018506ed743fbb
Firefox_Setup_52.9.0esr.exe	962a35b4b642c00519d1a3a87079e057
db-4.7.25.NC.zip	0582ef9de0cbc9d3ad89598ded6b56b5
boost-jam-3.1.17-1-ntx86.zip	72615486b39b0b6f5dfa91df531b7f7e
boost_1_37_0.7z	fcc70f33cf115f3d4b55e43073297b7c
WinCDEmu-4.1.exe	4e53befe779f677b1ccec54b84f60a8c
VS2008ExpressWithSP1ENUX1504728.iso	83b3292c1aa8e4d3eb0c6bcd66a55cd0
DebugView.zip	a937617b3dfebe3c6508f3eaad133e56

D Reproducibility

The empirical results reported in this work are supported by a reproducibility package available online. [1]

The repository is organized as a set of small, purpose-specific artifacts. Each directory corresponds to one experimentally isolated part of the pipeline and contains representative dumps, replay scripts, validation scripts, or structural decoders.

Directory	Verifies	Does not verify
`vlh/`	Offline validation of the kernel-side `pool` → `VeryLargeHashUpdate` → `seedbase_after` relation.	The provider-side derivation of `state20` from `seedbase_after` or from persistent provider state.
`fips/`	Replay of the final provider block H from observed `state20` and `aux20` to `out40`.	The upstream construction of `state20` or `aux20` from earlier provider state.
`provider/`	Local provider-side checks around the XOR-derived auxiliary input, FIPS entry material, and final copy behavior.	A complete offline reconstruction of the persistent provider state machine.
`rc4/`	Isolated validation artifact for the observed RC4 key-scheduling and byte-generation components.	The complete integration of RC4 into the provider state transition leading to `state20`.
`randwin/`	Structural decoding, sorting, and coherence checking of Windows-side `RAND_poll` input traces.	A replay of OpenSSL's internal PRNG state or a semantic decoding of every opaque byte blob.
`ssleay/`	Post-stir OpenSSL/SSLeay-side replay of the byte-generation path after OpenSSL's internal state has already been stirred.	The full OpenSSL stirring process from Windows and application-level inputs.
`wallet/`	Final wallet-level consistency check from an observed `RAND_bytes` output to the corresponding Bitcoin secret candidate, WIF encoding, public key, and P2PKH address.	Windows RNG internals, OpenSSL `RAND_poll`, or the provider-side `seedbase_after` to `state20` transition.

The package is intended to provide executable validation of the closed sub-components on representative campaigns. In particular, the provided artifacts allow independent verification of the following replay relations:

$$\texttt{pool} \rightarrow \texttt{VeryLargeHashUpdate} \rightarrow \texttt{seedbase\_after},$$

$$(\texttt{state20}, \texttt{aux20}) \rightarrow H \rightarrow \texttt{out40},$$

$$\texttt{out40}[0 : \texttt{0x20}] \rightarrow \texttt{CryptGenRandom}_{out}.$$

The `wallet/` component is a final consistency check. It shows that the observed 32-byte output from the OpenSSL `RAND_bytes` path can be interpreted as the corresponding Bitcoin secret candidate and that the derived WIF, public key, and P2PKH address values match the expected artifact. This check starts from an observed `RAND_bytes` output; it is not evidence about the internal Windows RNG, the OpenSSL `RAND_poll` stirring phase, or the provider-side transition from `seedbase_after` to `state20`.

1. https://github.com/Melik159/xp-cgr-replay

The `randwin/` component has a narrower role. It decodes, sorts, and checks the coherence of the Windows input records collected around the OpenSSL `RAND_poll` path, including Toolhelp snapshots, Lanman records, memory-status records, timing records, and opaque `CryptGenRandom` byte blobs. The `CryptGenRandom` blobs are treated as raw byte outputs, not as decoded Win32 structures. The `randwin/` component does not attempt to concatenate these inputs or replay the OpenSSL PRNG state transition.

The `ssleay/` component validates a different layer. It concerns the post-stir byte-generation behavior of OpenSSL's historical `ssleay_rand_bytes` path, after the internal OpenSSL state has already been stirred. It should therefore not be read as a reconstruction of the full OpenSSL seeding process from Windows inputs.

The reproducibility package deliberately separates structural inspection from cryptographic replay. In particular, it does not claim to reproduce the still-open provider transition

$$\texttt{seedbase\_after} \rightarrow \texttt{state20}.$$

This transition remains the main unresolved part of the full offline replay. The repository therefore supports byte-level validation of the closed sub-relations, while preserving the distinction between observed execution, offline-replayed components, final consistency checks, and the remaining provider-state derivation gap.

The historical Bitcoin client check is treated as an application-level consistency check rather than as a replay module. It verifies that the wallet artifact derived from the observed `RAND_bytes` output is accepted by an unmodified Bitcoin client and yields the expected address. It does not constitute additional evidence about the Windows RNG internals, the OpenSSL `RAND_poll` stirring phase, or the provider-side `seedbase_after` to `state20` transition.

For responsible artifact handling, the corresponding `wallet.dat` file is not distributed directly in the public repository, since it contains wallet key material even if generated in an isolated experimental setting. Researchers interested in reproducing the historical-client check may contact the author to obtain the artifact or reproduction instructions.