Table of Contents

Randomness of random in Cisco ASA

Ryad Benadjila and Arnaud Ebalard
ryad.benadjila@cryptoexperts.com
arnaud.ebalard@ssi.gouv.fr

ANSSI and CryptoExperts **

Abstract. It all started with ECDSA nonces and keys duplications in a large amount of X.509 certificates generated by Cisco ASA security gateways, detected through TLS campaigns analysis. After some statistics and blackbox keys recovery, it continued by analyzing multiple firmwares for those hardware devices and virtual appliances to unveil the root causes of these collisions. It ended up with *keygens* to recover RSA keys, ECDSA keys and signatures nonces. The current article describes our journey understanding Cisco ASA randomness issues through years, leading to CVE-2023-20107 [2,6]. More generally, it also provides technical and practical feedback on what can and cannot be done regarding entropy sources in association with DRBGs and other random processing mechanisms.

1 Introduction

1.1 Use of randomness in network devices

Random numbers are used in multiple aspects of network equipments operations, all the more so when those are dedicated to security tasks:

- For administration interfaces or VPN services for users, random numbers are the root of protocols like TLS and IKE : even if state of the art primitives like AES encryption or ECDSA signature are used inside state of the art protocols, failure at providing correct entropy to those primitives may result in catastrophic failure of security product functions.
- To defend against exploitation of vulnerable code, mechanisms like ASLR rely on (obviously) non predictable random values.
- More generally, high level protocols expect non-predictable session identifiers, tokens, or nonces.

A difficult aspect with randomness that must be taken into account is that it is difficult (if ever possible) to get definitive proof of practical quality of produced output over time.

** Work performed while at ANSSI.

1.2 Prior art and related work

The best way to understand practical randomness issues, even when using state of the art theoretical primitives, is to consider previous failures of such primitives in other systems.

Without reducing the subject to the list below, the following examples can serve as showcases to grasp both possible root causes as well as catastrophic impacts of randomness failures.

In 2008, a small change in Debian OpenSSL package resulted in predictable RNG operations [16], itself leading to the generation of broken SSH, SSL RSA and DSA keys ... worldwide.

In 2010, nonce duplication in Sony PlayStation 3 code led to the recovery of the firmware signature private key [8], allowing unpatchable jailbreaks of the device on the existing consoles.

In 2012, various embedded devices with low boot entropy generated duplicated primes during RSA key generations [11], leading to trivial private key recovery from the set of public keys using a simple GCD algorithm.

In 2013, nonce duplication in ECDSA signatures in the Bitcoin blockchain allowed recovery of associated ECDSA private keys [22], then allowing access to the fund at those addresses.

In 2019, Cisco published a reported vulnerability resulting in the generation of low entropy keys on their ASA and FTD software based devices [10].

The current article details an independent rediscovery of this vulnerability with interesting twists and a deeper understanding of its root causes, as well as the exposition of more vulnerable devices and more vulnerable certificates in the wild. Indeed, [10] neither details the origins nor the impacts of the (barely mentioned) entropy issues, which lead to keygenning in many cases as we will present in the current article.

For french-capable readers, [19] might provide a good synthesis of the topic.

1.3 Reading path

The structure of the document globally follows the logical and chronological way the study was performed. As it all started with the discovery of duplications on public certificates, section 2 provides a thorough analysis of those ECDSA and RSA certificates, the discovered issues and impacts and possible key recovery from such a set. To get to the root cause of the issues, an understanding of Cisco ASA ecosystem, hardware devices,

virtualization and debug capabilities, firmware content, etc. was required. These elements are covered in Section 3. Section 4 provides reminders on RNG topics, with minimum required information on DRBGs (CTR-DRBG and Hash-DRBG), OpenSSL MD_RAND and also some analysis of implementations/mechanisms found in Cisco products, like RSA Labs BSAFE. Building on those two sections, section 5 provides an external analysis of randomness issues for certificates generated on ASAv virtualized appliances and goes to the origin of those issues, using a specific version as a support for this work. All keygened versions are also covered with less details in this section. Section 6 builds upon this analysis and specificities of hardware ASA devices to speculate on the root causes of observed duplications on those platforms. Finally, the conclusion tries to summarize all the lessons learned during this journey on practical RNG implementations, and provides some advice for developers.

1.4 Disclaimer

This article takes as basis the randomness issues impacting the generation of self-signed certificates on Cisco ASA products to get to the root causes of those specific issues on those platforms. The other possible impacts of low randomness on those platforms (aside from certificates generation) are not analyzed; the main purpose of this work being to alert other developers and the community regarding the use of low randomness sources. Namely, other cryptographic material such as IKE and SSH keys might (or might not) be impacted with more or less severity, but this is not the subject of the current article.

2 External observations and initial recovery

2.1 ECDSA certificates

While developing [31] and [3], the idea arose to perform some tests on `r` component of ECDSA signatures in our large dataset of X.509 certificates. This set, built with the years from public sources for test purposes, contains more than 250 millions unique X.509 certificates.

The extraction and search for duplication of the `r` components of the signature in ECDSA-signed certificates from the set resulted in an non-empty list. Because `r` is computed in the following way during signature process, collisions can only happen due to a random generator issue; this generator returning the same value `k` twice during different certificate signing operations as presented on Algorithm 1.

1: Get a random value k in]0, q[
2: Compute $W = (W_x, W_y) = k \times G$
3: Compute $r = W_x \ mod \ q$
4: ...
5: Return (r, s)

Algorithm 1. Generation of `r` during ECDSA signature process; G being the group base point on the curve and q the order of the curve.

An analysis of the impacted certificates quickly showed that they were all very similar in their structure as exhibited in Figure 1.

```
Data:
    Version: 3 (0x2)
    Serial Number: -2145020325 (-0x7fda69a5)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = ASA Temporary Self Signed Certificate
    Validity
        Not Before: Sep 10 08:04:15 2018 GMT
        Not After : Sep  7 08:04:15 2028 GMT                  -- 10 years
    Subject: CN = ASA Temporary Self Signed Certificate
    Subject Public Key Info:
        Public Key Algorithm: id-ecPublicKey
            Public-Key: (256 bit)
            pub:
                04:76:52:e0:cf:12:4c:11:22:e6:da:75:53:09:97:
                5c:fa:1f:4e:b3:dd:8e:42:65:28:2d:59:52:18:e9:
                b3:04:d5:3c:a6:b3:6f:a5:ee:01:2c:91:c4:e1:fd:
                bb:cb:52:60:3e:f2:8b:ae:c1:42:1f:76:57:28:64:
                d6:48:e6:c3:c3
            ASN1 OID: prime256v1
            NIST CURVE: P-256
Signature Algorithm: ecdsa-with-SHA256
     30:45:02:20:36:76:d5:e1:2e:62:91:db:7d:28:6f:ed:fa:fd:   -- r
     15:5f:3e:4f:fb:4c:9b:f8:79:c7:dd:ba:0d:19:1d:80:27:18:
     02:21:00:fd:a7:81:76:c6:da:22:30:82:09:b8:dc:c9:38:ad:   -- s
     94:6e:72:b4:14:63:65:88:63:b4:f7:86:d7:17:53:f8:ed
```

Fig. 1. Cisco ASA self-signed ECDSA certificate

Some statistics We extracted from our set all the certificates matching this template (ECDSA, same very specific Common Name `CN`, etc.) to end up with a subset of 313k ASA ECDSA self-signed certificates.

The statistics regarding `r` duplication in this subset went:

— $\approx 82k$ certs with a duplicated `r`, *i.e.* 26.4% of the set.

— $\approx 18k$ `r` appear between 2 and ... 44 times!

Expecting self-signed certificates of the subset to embed non-colliding public keys, we also did some statistics on the topic:

— $\approx 113k$ certs with a duplicated pub key, *i.e.* 36.1% of the set.
— $\approx 16k$ pub keys appear between 2 and ... 704 times!

Keys and nonces recovery A strong hypothesis for the resistance of ECDSA signature mechanism is that the nonces `k` are fresh random values uniformly sampled from the set of positive integers smaller than the order of the base point of the EC parameters. Considering the length of `k`, a repetition of a `k` value never happens when a decent RNG is used. It is well-known that a repetition of a nonce value can be catastrophic for the security since the private key can then be recovered from the signed messages and their signatures [8].

For a pair of signatures (r_1, s_1) and (r_2, s_2) of different messages m1 and m2 with a colliding nonce `k`, we have $r1 = r2$, which we note r. A simplified version of ECDSA signature mechanism is presented on the left part of Figure 2 along with nonce recovery mechanisms on the right part of the same Figure.

1: h = H(m)
2: e = OS2I(h) mod q
3: $k \Leftarrow$ R, $k \in]0, q[$
4: $W = (W_x, W_y) = k \times G$
5: $r = W_x \ mod\ q$
6: $s = k^{-1} \times (x \times r + e) \ mod\ q$
7: Return (r, s)

From 6. on the left side, we draw:

$$\begin{aligned}(s_1 - s_2) &= k^{-1} \times (xr + e_1) - k^{-1} \times (xr + e_2) \ mod\ q \\ &= k^{-1} \times (xr + e_1 - xr - e_2) \ mod\ q \\ &= k^{-1} \times (e_1 - e_2) \ mod\ q\end{aligned}$$

$$\begin{aligned}\Longrightarrow k &= (e_1 - e_2) \times (s_1 - s_2)^{-1} \ mod\ q \\ \Longrightarrow x &= (k \times s_1 - e_1) \times r_1^{-1} \ mod\ q \\ &= (k \times s_2 - e_2) \times r_2^{-1} \ mod\ q\end{aligned}$$

Fig. 2. (Simplified) ECDSA Signature mechanism and duplicated nonce recovery equations (for nonce and private key)

Considering the length of `k`,[1] this can simply never happen with a decent RNG. Nonetheless, if such an event occurs (which can be spotted with identical `r` values), this allows for a trivial recovery of the private key.

Additionally, mathematical computations involved in ECDSA signature process also allows the private key owner to recover the nonce `k` associated with `r` from the signature and the signed message. Getting access to all

[1] 256 bits in Cisco ASA case, *i.e.* the size of the order `q`.

the nonces ever used by a signer is usually useless ... except when the `r` values resulting from those nonces are duplicated in other signatures performed with differents keys.[2] This is what happens in our set.

Having certificates signed with the same nonce and embedding the same public key provides trivial access to that private key. Having access to that key provides access to all the nonces for the certificates signed with this key. It then provides access to the keys of all certificates whose `r` value is associated with one of these nonces. This results in a trivial iterative converging Algorithm 2 for private key recovery in our subset, providing the results in Figure 3.

Starting with 737 recovered keys ($\approx 3.7\%$ of the $200k$ certificates with unique keys) because of duplicated nonces, this iterative process allows in a few steps a final recovery of 4739 keys ($\approx 23.7\%$) from this subset.

Input: A_{pubk} set of all public keys, A_r set of all r (from the certificates)
Output: S_{privk} set of recovered private keys, S_k set of recovered nonces
1: Get all keys from A_{pubk} used with duplicated nonces from A_r
2: Apply algorithm on Figure 2 to recover a set of private keys S_{privk} and a set of nonces S_k
3: From recovered S_k break unknown private keys from A_{pubk} where the recovered nonces are used and inflate S_{privk}
4: From recovered S_{privk} break unknown nonces from A_r where the recovered private keys are used and inflate S_k
5: Did we reach a point where S_{privk} and S_k did not inflate in the last two steps? If yes **exit**, else **goto step 3**

Algorithm 2. (Simplified) Iterative keys and nonces recovery algorithm

CVE-2019-1715 The result presented on Figure 4 led us to contact Cisco PSIRT through CERT-FR to report the issue on 10 Feb 2022, who related this to CVE-2019-1715 [10] that indeed had an explicit title: *Low-Entropy Keys Vulnerability*.

At that point, even if our findings matched this CVE, the huge number and percentage of impacted certificates observed on Figure 4, along with the advertised issuance dates[3] in the subset - as presented below - led

[2] Actually, knowing even a small fraction of the nonce bits can lead to a HNP (Hidden Number Problem) [33] that can break the private key with a set of known signatures. We are not in this case as it will be unveiled in the sequel, hence we only focus on the full duplication of nonces.

[3] The value of `notBefore` field.

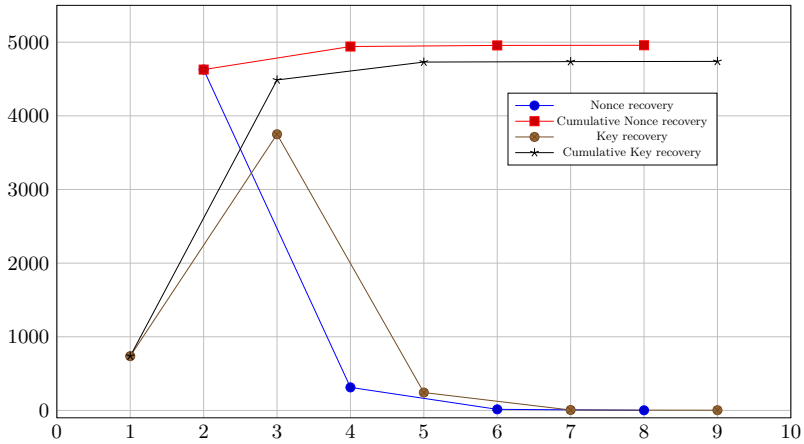

Fig. 3. Results of iterative keys and nonces recovery on the subset

us to consider that either some products had been missed, or the fix was failing, or a huge amount of users just had not fixed the issue.

As our dataset did not have data after 2021, we have also checked the publicly available Rapid7 [28] Open Data databases and expanded our total ASA ECDSA certificates number from 330k to 540k with `notBefore` lying in 2022. We have confirmed on the Rapid7 only dataset as well as on the merged one that the obviously weak ASA certificates proportion remains the same, even in 2022, with what we have observed in the original set.[4] We have also checked on Shodan [32] that hundreds to thousands of currently alive machines have obviously weak certificates.

2.2 RSA certificates

During the extraction of Cisco ASA certificates based on their `CN` from our large set, the resulting subset contained both ECDSA and RSA self-signed certificates. The existence of both RSA and ECDSA certificates in the subset is explained by the fact that Cisco ASA products generate both kinds of certificates at boot.

Expecting RNG issues on the platforms to also impact the generation of RSA keys, we first started performing GCD between RSA modulus N ($= P \times Q$) extracted from those certificates, expecting duplicated P values allowing to recover Q values as covered in [11].

[4] We want to thank our colleagues from the LED laboratory and SDO entity at ANSSI for helping us with the Rapid7 data extraction and exploitation.

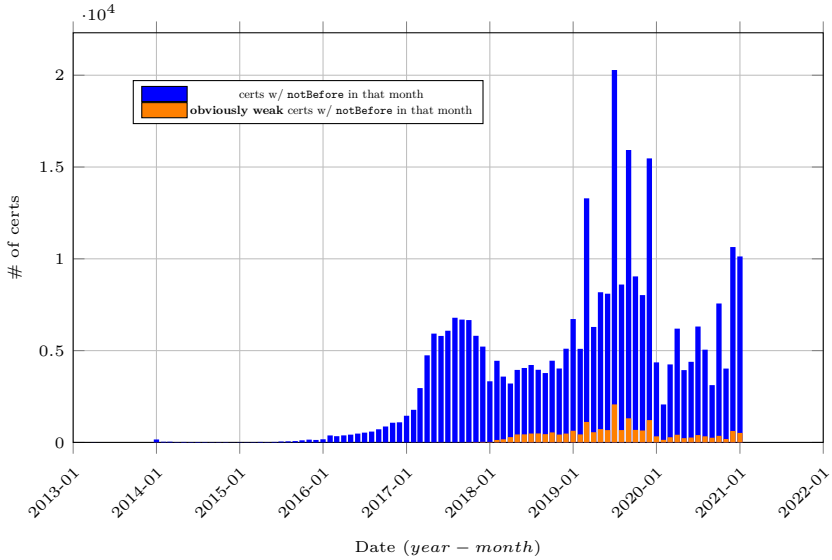

Fig. 4. Number of Cisco ASA ECDSA certificates per month (found vs obviously weak)

From the RSA subset of 200,466 certificates at hand, we found no GCD issues. Wondering why RSA key generation seemed not to be impacted by RNG issues, we started considering we missed something more obvious. We decided to look for trivial modulus duplications which provided positive results. A summary of the results are given in Figures 5 and 6.

The result of this initial analysis shows that between 6 and 10 % of Cisco ASA self-signed RSA certificates in the wild contain duplicated modulus. This explains why previous GCD checks did not provide any positive result: modulus with a common P also had the same Q.

Among our 200k Cisco ASA RSA self-signed certificates, we found a total amount of 12,226 certificates with a duplicated modulus. This represents 2,194 modules which appear more than once in the certificates set, with a repetition count between 2 and 2,492. Based on the advertised generation date found in `notBefore` field in the certificate, the issue seems to have begun between 2016 and 2017, and continues with no decrease long after the publication of the CVE-2019-1715 [10] and associated fixed software by Cisco, with obviously weak certificates still appearing until the end of our data set in 2021.

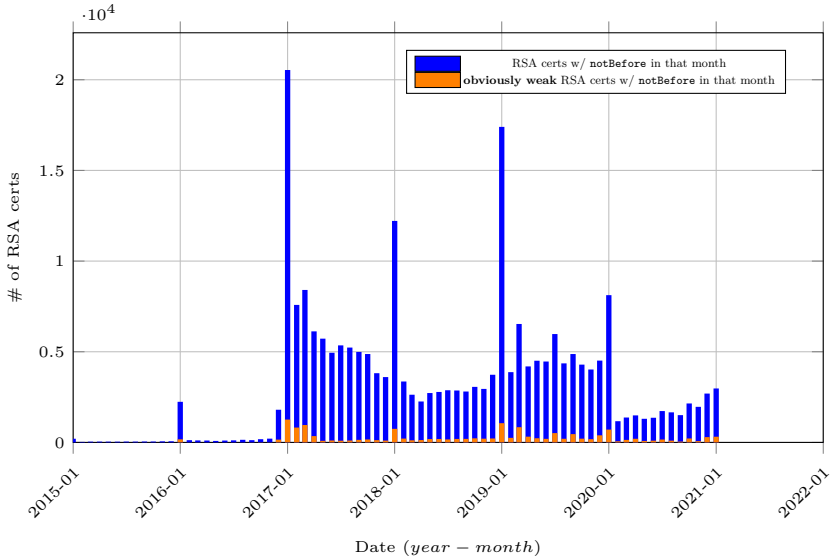

Fig. 5. Number of Cisco ASA RSA certificates per month (found vs obviously weak)

3 ASA devices

3.1 Hardware devices and ASAv virtual appliances

The Cisco ASA family of products comes in two main flavours: hardware appliances as well as ASAv virtual appliances (although these two tend to converge towards a unique platform in the recent years). Hardware appliances are made of dedicated hardware with network acceleration chips and have historically been seen as the main selling devices for professional deployment. Virtual appliances appeared a few years ago as interesting alternatives for testing purposes, *e.g.* on network simulation platforms such as GNS3 or EVE-NG [7, 9], and bringing more flexibility and scalability in VPN deployment scenarios. With the democratization of cloud solutions, and the easy scaling of such virtual appliances, Cisco seem to put efforts on developping the ASAv virtualization based solutions. More and more hardware appliances in fact embed hardware capable Xeon CPUs with ASAv, which is cost effective from a deployment and development standpoint.

The "classical" hardware appliances are made of a main CPU based on a `x86` 32-bit or 64-bit architecture (low power AMD Geode or Intel Atom for low-end, more capable AMD Opteron or Intel Xeon for high-end). A companion cryptographic accelerator on the PCI bus offloads the

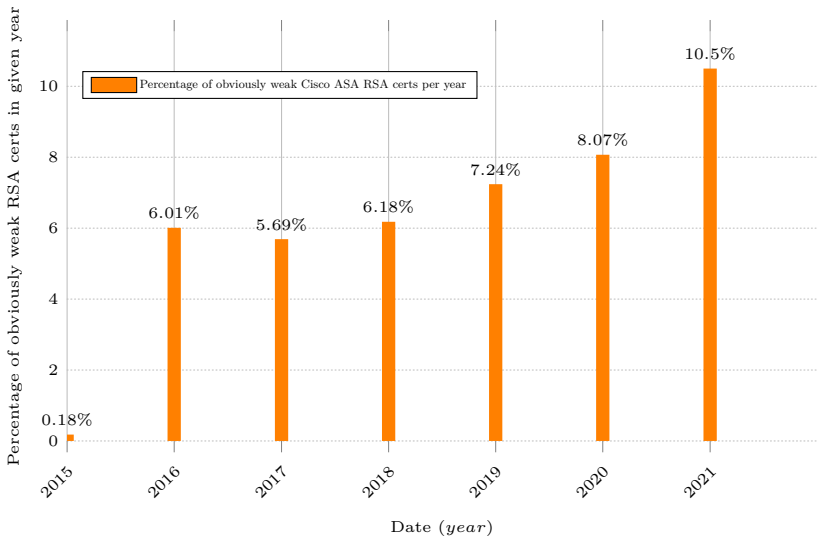

Fig. 6. Percentage of obviously weak RSA certificates per year

heavy network load and cryptographic operations (for TLS, IKE, etc.). This accelerator comes in the form of a Cavium IP (Octeon or Nitrox Lite for low to middle range, Nitrox PX for high-end). Figure 7 provides an overview of the main CPU and Cavium found in current Cisco ASA devices.

ASAv are usually provided in the form of flat files for virtualization solutions (`qcow2`, `ova` and so on). The ASAv packages are hence self-contained and are expected to run on most of the hypervisors (`kvm`, `hyperv`, etc.) or bare-metal, *e.g.* using a `x86` emulator. For both ASAv and hardware solutions, a licensing model is used to configure functionalities and remove forced restrictions on the same platform.

During our work, **we have chosen to focus on the specific ASA hardware device 5506-X** as it was both still supported by Cisco, cheap ($\approx$ 40€ on a classified ads website for a brand new device) and easily available. This explains why **we only present statistics on this platform in the sequel**. We emphasize however that our results could be extended to (at least some) other hardware platforms as they share a very similar configuration, as can be seen on Figure 7, and use common binaries in the firmware.

ASA Device	Crypto Accel	CPU
5505	Cavium Nitrox Lite CN505	AMD Geode LX 800 500MHz
5506-X	Cavium Octeon III CN7020	Atom C2000 series 1250 MHz
5506W-X	Cavium Octeon III CN7130	Atom C2000 series 1250 MHz
5506H-X	Cavium Octeon III CN7130	Atom C2000 series 1250 MHz
5508-X	Cavium Octeon III CN7130	Atom C2000 series 2000 MHz
5510	Cavium Nitrox Lite CN1010	Pentium 4 Celeron 1600 MHz
5512-X	Cavium Nitrox PX CN1610	Clarkdale 2793 MHz
5515-X	Cavium Nitrox PX CN1610	Clarkdale 3059 MHz
5516-X	Cavium Octeon III CN7130	Atom C2000 series 2400 MHz
5520	Cavium Nitrox Lite CN1010	Pentium 4 Celeron 2000 MHz
5525-X	Cavium Nitrox PX CN1610	Lynnfield 2394 MHz
5540	Cavium Nitrox Lite CN1010	Pentium 4 2000 MHz
5545-X	Cavium Nitrox PX CN1610/20	Lynnfield 2660 MHz
5550	Cavium Nitrox Lite CN1010	Pentium 4 3000 MHz
5555-X	Cavium Nitrox PX CN1620	Lynnfield 2792 MHz
5580-20	Cavium Nitrox PX CN1520	AMD Opteron 2600 MHz
5580-40	Cavium Nitrox PX CN1520	AMD Opteron 2600 MHz
5585-X SSP-10	Cavium Nitrox PX CN1620	Xeon 5500 series 2000 Mhz
5585-X SSP-10 EP	Cavium Nitrox PX CN1620	Xeon 5500 series 2000 Mhz
5585-X SSP-20	Cavium Nitrox PX CN1620	Xeon 5500 series 2133 MHz
5585-X SSP-20 EP	Cavium Nitrox PX CN1620	Xeon 5500 series 2133 MHz
5585-X SSP-40	Cavium Nitrox PX CN1620	Xeon 5500 series 2133 MHz
5585-X SSP-60	Cavium Nitrox PX CN1620	Xeon 5600 series 2400 MHz

Fig. 7. Processors/Cavium accelerators in ASA devices (source: `https://community.cisco.com/t5/security-blogs/asa-and-firepower-hardware-fact-sheet/ba-p/3665136`). Highlighted device is the one acquired for our experiments

3.2 Firmware content, execution and analysis

We provide hereafter a brief overview of the ASA firmware ecosystem: we are not exhaustive as we only focus on the information needed for the sequel of this article. For a detailed tour of taxonomy and history of ASA devices, please refer to the comprehensive NCC group blog posts [20]. An interesting thing to notice is that both hardware and ASAv platforms share the same core executed binary: a flat `.bin` file which contains an embedded Linux kernel and its `rootfs`, only the bootloader differs. `grub` is used for ASAv and the proprietary `rommon` for hardware solutions. The platform (`init` scripts after the kernel has booted) detects the running environment and adapts some configuration depending on it. For instance, on a hardware platform the Cavium dedicated firmware will be pushed through the PCI bus, and proprietary communication will be established between the main CPU and the Cavium afterwards. On ASAv, the running environment is aware that no cryptographic accelerator is present and everything is performed on the main CPU. Also, as we will see, various

code paths can be executed depending on the main CPU capabilities (*e.g.* AES or SHA-2 acceleration instructions, `rdrand`, etc.).

For ASAv, the `.bin` is packaged with the `grub` bootloader as well as another partition emulating the flash (the `rootfs` being read-only, the "flash" is used for configuration and state data), and these elements form the flat file to be virtualized, *e.g.* as an `ova` file.

For hardware platforms, the `.bin` was historically directly loaded by `rommon`, and the flash device is a physical hardware device in the form of Compact Flash (CF) card or embedded eUSB flash chip. The `.bin` is on this device and the "user" configuration data is on a dedicated partition. `rommon` reads the flash, performs some checks and boots the `.bin`. It is also possible to load the `.bin` file using `tftp` or an USB thumb drive connected to the appliance.

In recent hardware platforms (since around 2018 with the release of FTD FirePOWER integration in ASA), firmware files are now `.SPA` files. This file format is also concurrent with the usage of **secure boot** technologies in Cisco products: modern hardware releases now integrate UEFI with secure boot, a dedicated ACT chip against counterfeiting as well as a custom FPGA (the Cisco Trust Anchor module for software checking) are used as roots of trust to ensure that the loaded firmware is authentic (signed) and not tampered with. The `rommon` bootloader is implemented as an UEFI module and extends the trusted anchor of UEFI secure boot, with additional work from the FPGA and the ACT chip. Although the exact process of secure boot is not described by Cisco, some elements can be found in [14]. Additionally to the kernel and `rootfs`, `.SPA` files also contain cryptographic metadata that help the secure boot flow and the authentication of the image. It is also worth noticing that both for `.bin` and `.SPA` format, similar extensions are used for `rommon` as well as other modules updates (such as other UEFI modules, the FPGA bitstream, etc.) with authenticity checks whenever necessary when secure boot is supported.

Once the boot process is finished in both ASAv and hardware platforms, the Linux kernel launches a binary called `lina_monitor` whose task is to execute the main executable with dedicated options: `lina`. It is a monolithic binary whose size has been increasing over the firmware versions (from around 100 MB in the 9.8 versions to around 180 MB in recent 9.17 versions). The binary has elevated privileges and embeds all the functionalities that are expected from the platform: drive the network interfaces, interact with the cryptographic accelerator if present, monitor the state and health of the running environment, etc. It implements

various abstraction layers inherited from old Cisco non-Linux IOS era, which might explain the monolithic approach, whose purpose is portability across existing Cisco platforms. This explains why thousands (more than 110k) of functions are present, with lots of aggregated libraries in different versions (*e.g.* 30 instances of SHA-2), with or without customization by Cisco (*e.g.* OpenSSL is mostly genuine except from some low-level parts discussed in the next sections). Analyzing all the possible execution paths for all the possible platform with static analysis is made very complex by all the abstraction layers and the combinatorial complexity. The `lina` binary is stripped, but debugging strings used in `printf` like functions ease the static analysis.

3.3 Instrumentation

There is a substantial previous work on Cisco ASA jailbreaking and instrumentation. For the sake of brevity, we advise the curious reader to refer to the extensive work of NCC group [21] that builds upon (and details) a long history of CVEs in their articles. We have based a large part of our analysis and our instrumentation on their tools by modifying their `asafw` and `asadbg` frameworks for our specific purposes. They explain how to modify the firmware images in order to inject a `gdbserver` binary and disable the various protections that exist, namely: integrity checking of the `lina` binary by the `lina_monitor` launcher, ASLR (Address Space Layout Randomization), the watchdog in the `lina` binary itself that sends termination signals whenever there is a stall (*e.g.* with breakpoints), etc.

Although at the time of release, NCC group's tools allowed to instrument both ASAv virtual images as well as hardware images (*e.g.* for 5505 or 5512-X), recent hardware releases (of interest for us) such as the 5506-X make use of **secure boot** technologies as previously described. Secure boot limited our instrumentation of the available 5506-X hardware since we could not inject our `gdbserver` and modify the image without being detected by `rommon`. Recent attacks on all ASA firmware presented at BlackHat [13] would have allowed a tethered jailbreak (*i.e.* inject the instrumentation payload post-boot with an exploit providing root privileges access). However, we did not explore this path because these exploits appeared late during our work on ASA and we lacked time to integrate and use them. Moreover, we suspect that some of the reasons behind the entropy fragility also lies in the Cavium firmware behavior: beyond the mere instrumentation of the main `x86` CPU code, instrumenting the cryptographic accelerator is another topic that would require a

substantial additional work (unveiling Cisco's proprietary protocol for the communication over PCI between `lina` and the MIPS processor).

Hence, we will hereafter describe our gray box dynamic instrumentation on ASAv that allowed us to capture the executed code paths, understand the flaws and implement key gen tools for various firmware versions. We will then describe the pure black-box methodology we used on the 5506-X to expose the fragility of some firmware versions in their hardware fashion: because of the secure boot limitation, the analysis is limited and these weaknesses are not exploited (beyond nonce reuse issues previously described).

5506-X instrumentation Since secure boot prevents exploitable modification of 5506-X firmwares, we have chosen a pure black-box approach. We have implemented a Python script that communicates with the hardware appliance through the serial console as well as through the network using the `expect` module to automate our commands.

The serial console is mainly used to drive the `rommon` bootloader and boot using `tftp` a target firmware. Then, the firmware is configured with minimal elements to have the network and generate the RSA and ECDSA certificates during boot. Then, a TLS session is opened using the SSL Python module, and recovers these certificates (with proper ciphersuite downgrading whenever necessary to recover the RSA certificate).

Since we know that the absolute time of the appliance can play a role in randomness generation, we explicitly set the same time during reboot. Although some jitter of a few seconds exist (because of non deterministic boot time), getting many samples allows us to get certificates generation time collisions (in seconds, that we validate using `notBefore`), which exhibit interesting results as we will expose in the next sections.

ASAv instrumentation Our dynamic instrumentation of the ASAv firmware versions used three complementary aspects:

— Modify NCC group's `asafw` and `asadbg` scripts to adapt to various recent versions of ASAv firmware (*e.g.* handle ASLR in different fashions, properly repacking when injecting new binaries in all situations, etc.). We also decided to get rid of the GNS3 network simulation framework for instrumentation, as it added too much complexity for little useful features in our context: we used custom scripts to handle virtual interfaces and bridges creation, and a dedicated `qemu` command line. We have also developed a Python script

based on the `expect` module to automatically perform the appliance configuration on the (virtual) serial line. This Python script is very similar to the one described in the 5506-X instrumentation.
— Develop dedicated `gdb` scripts that allow to dynamically sample values from memory at breakpoints, inject data at will, and get a better view and big picture of the code paths. These scripts became quite complex as they could execute up to 30 breakpoints, and they behaved quite well for most of the firmware versions, showing that this methodology scales well even for a multi-threaded large binary. Among other things the script dumps the RSA and ECDSA certificates in DER format after their generation.
— Develop dedicated `qemu` plugins to push the instrumentation further. As we will explain hereafter, although `gdb` is a very powerful tool for dynamic analysis, some situations required a less "invasive" monitoring of the `lina` binary in action (ideally a pure black-box observation of the untouched binary).

As it will be highlighted in the dedicated Sections 4 and 5, some versions of the ASAv firmware use absolute time and relative timings of events as entropy sources or additional data. In such cases, using breakpoints with `gdb` will completely break these timings: it is not possible to observe key generations with "real" values (*i.e.* values that the untouched binary would produce) since breakpoints use delays with too much volatility for the target precision. Another issue arising with `gdb` instrumentation is the observation of the values of initialized or uninitialized input buffers when these are used as an entropy source (*e.g.* with the `MD_RAND` processing): the multi-threading nature of `lina` makes catching these buffers actual values hard with a sheer debugging approach. Even harder elements to capture are when these buffers contain ASLR chunks that are random, but that do not appear so under `gdb` scrutiny as ASLR must be deactivated for proper debugging of `lina`.

For all such cases, a pure black-box approach that does not tamper with the running `lina` memory map and behaviour is needed, and `qemu` emulation mode is perfect for this. For a quick and efficient instrumentation of the emulation, we have chosen to develop a TCG plugin based on the provided `tests/plugins/insn.c` and `contrib/plugins/execlog.c` examples [27]. The idea is to be able to sample or modify the interesting instructions and dump memory (the timings we need to sample, input buffers raw content, etc.) at will while freezing the whole firmware. We had to face some challenges that we will briefly describe hereafter:

— Recovering and modifying memory content from the TCG plugins is not straightforward as these plugins are not dedicated for this (especially when we want to modify elements such as registers or memory as these plugins are mainly for read-only instrumentation). We have diverted the `qemu gdb` debugging stubs (that already contain low-level memory access functions) to perform this.
— We had to deal with ASLR in our `qemu` instrumentation. Indeed, when we want to observe a given address in the binary (taken from a disassembler), this address will not be the same when running. We have implemented some filtering heuristics that use the fact that ASLR does not modify the lower 12 bits page offset of the `rip` instruction pointer. Using these 12 bits with the pointed instruction opcode allows for an efficient disambiguation.
— Since TCG plugins inherently depend on the basic translation blocks of `qemu`, there could be some desynchronization between the sampled values (with our custom read/write stubs) for the registers and their real values at a given point of the program. Resynchronization points are the end of translation blocks, usually corresponding to the end of basic blocks. We took advantage of registers values preservation across instructions (or copy of these values in other registers) to get the job done in the different situations we had to face.

All-in-all, the `qemu` instrumentation allowed us to confirm our analysis of the timings in the CTR-DRBG cases, confirm our analysis of the Hash-DRBG behaviour, and confirm static (initialized) buffer values as well as ASLR feeding values to MD_RAND (see Section 4 for more details about these RNG engines).

4 Cryptographic and randomness players

In this section, we describe the primitives that contribute to the randomness generation as we analyzed them from the `lina` binary using static and dynamic analysis through instrumentation on ASAv versions ranging from 9.6 to 9.10. We focus on brevity for the sake of readability, although the analysis was a winding road with non trivial call graphs and arduous `gdb` scripting. The purpose is to have an overview of full key and random generation paths from the entropy sources to the output, which will help understanding their fragility and how keygenning might be possible for some of the generated key material.

We classify the primitives that are needed for the understanding of key/nonce generation in three layers, from upper to lower ones:

- the top ECDSA and RSA material generation;
- the deterministic RNG engines producing this material;
- the entropy sources and lifting routines that feed the engines.

It is to be noted that the engines can also be fed with other deterministic engines that use entropy sources, with as many layers as needed. Actually, things are more complicated than what is summarized here: various backends can be used at runtime, some are selected when the `lina` binary starts, others are seen as fallback of failing engines, switches between engines can occur during the course of the execution etc. This becomes even more crooked when entropy sources and lifters that feed the engines also go through such runtime choices. This leads to a very complex call graph and many possibilities that we certainly do not pretend to exhaustively comprehend by static analysis (and even with dynamic instrumentation). However, we have tried to limit the analysis complexity by only focusing on the (boot time) certificates generation. We have also observed that for a given ASA firmware version, the same execution paths are taken for random generation during nominal executions (*i.e.* the randomness generation call graph is kind of deterministic in early boot). Hence, this section only bears down on the cryptographic primitives and RNG players that are of interest.

All the results provided here have been validated using post analysis and instrumentation with black-box checking and/or non-invasive `qemu` values sampling. As a disclaimer, we have only focused on the random generation paths that are executed in ASAv using our virtualization environment (`qemu` with `kvm` on laptops with Intel CPUs). We have chosen as the virtualized CPU an *Intel Nehalem* that we thought representative of average deployed ASA appliances. This CPU microarchitecture does not embed the latest technologies such as `rdrand`, and as it will be discussed this can make a difference when it comes to entropy sources (shifting from a very fragile random generation to a somewhat more robust one). Although this might not be exhaustive from an analysis point of view, this has been enough to produce keys found in our certificates dataset which proves at least that these paths are executed on existing platforms connected to the internet.

4.1 RSA and ECDSA generation process

The key and nonce generation processes all make use of randomness pulled from instantiated random generators backends that will be described

in the next section (deterministic generators). For this section, consider them as random bytes providers. Figure 8 and Figure 9 provide a high level view of when key material is generated. Entropy lifters are used to feed some deterministic generator `Instantiate()` and `Generate()` functions. The `Instantiate()` routine is called first to initialize the random generator, then each time random bytes are needed the `Generate()` routine is invoked: Figure 8 shows how chunks of 16 bytes, then 3×16 (four times), and then 28, 32 and 8 bytes are pulled from this generation process. The RSA modulus and the ECDSA private key / nonce are generated using some of these extracted bytes. The RSA modulus is generated first, then the ECDSA private key and finally the nonce (which respects the most natural way of performing the operation).

RSA prime factors and modulus The RSA prime factors follow a seeded generation as described in FIPS 186-4 method B.3.4 [24]: using 28 random bytes, two 1024 primes P and Q are generated and the modulus $N = P \times Q$ is computed. This perfectly explains why the GCD attacks did not work on the certificates dataset: the collisions are on the modulus N but no two distincts modulus share a common factor.

ECDSA private key The ECDSA `secp256` private key is generated at top level using OpenSSL original generation procedure taking 32 random bytes and reducing them modulo the order of the generator.[5] The ECDSA private key being generated after the RSA modulus (hence with more randomness in the random engine), less collisions are expected for them than for RSA keys (which we indeed observe on our dataset).

ECDSA nonce We have discovered that the ECDSA `secp256` nonce can be generated in at least two ways depending on the considered firmware and running environment. For a puzzling reason, none of them uses the original OpenSSL nonce generation primitive. The first generation procedure makes use of the RSA Labs BSAFE library [29] [6]: two random samples, of 16 bytes and 8 bytes, are used to feed proprietary BSAFE post-processing engines based on SHA-1 and modular reductions to produce 32 bytes. The only important takeaway is that they are deterministic with no other inputs (in this case, the nonce also has a theoretical entropy of 24 bytes

[5] `BN_rand()` in `EC_KEY_generate_key()`

[6] Which is a pain to analyze, since we switch from OpenSSL clear APIs to proprietary big numbers abstraction layers

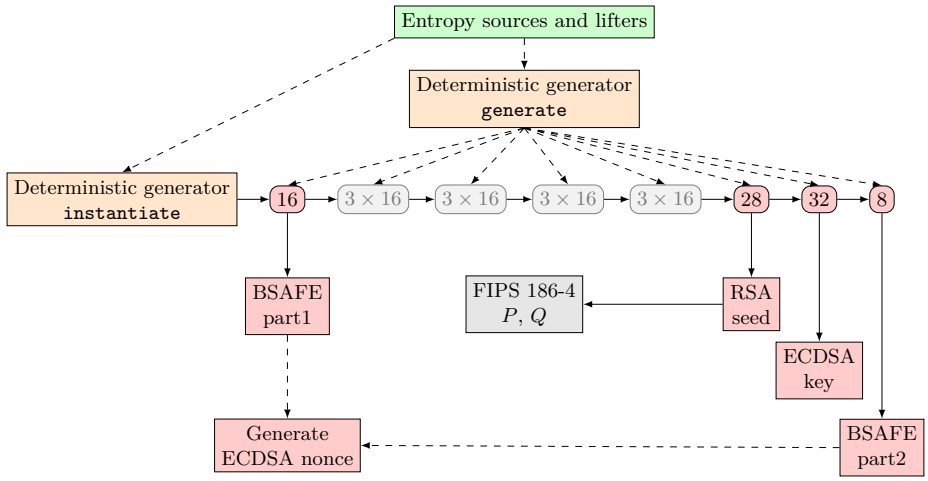

Fig. 8. RSA modulus and ECDSA private key/nonce generation

instead of 32, which is bad but less catastrophic than what will be exposed later). The second generation procedure directly samples 32 bytes from another deterministic random backend which is a Hash-DRBG (more on this later). Finally, the produced 32 bytes are reduced to generate the nonce. For the same reasons we have less collisions for ECDSA keys than for RSA modulus, we expect from the first generation procedure to have even less collision for nonces.

Keys relations As we have explained, RSA and ECDSA key material is generated sampling from a deterministic random generator. This means for example that if we are able to find the RSA 28 bytes seed, we get an output from the generator. If it is possible from these 28 bytes to guess the next 32 bytes with few additional efforts, the ECDSA key is obtained! Also, the ECDSA key and nonce are related (if we get one we get the other using an equation). The next sections are about the possibilities of exploiting these "steppings" forward or backward in the random generation flow to guess unknown value with much less complexity than the exhaustive search. For this, we still have to understand the deterministic generation producing bytes at each step, as well as the entropy sources that feed this generation: this is the purpose of the next descriptions.

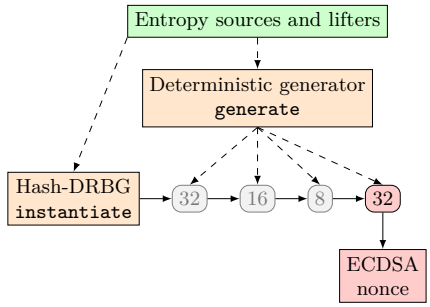

Fig. 9. Alternative ECDSA nonce generation

4.2 Deterministic generators

We present hereafter the deterministic engines that are used to produce the random bytes consumed by the upper layers in `lina`. As we have already stated, this is not an exhaustive enumeration: we only exhibit the engines used for our purpose (*i.e.* RSA and ECDSA keys and certificates generation). They are of two types: NIST DRBGs and MD_RAND. In the sequel, we will use equivalent nomenclatures for these deterministic engines: RNG or PRNG for (Pseudo) Random Number Generators.

An interesting thing to notice when analyzing `lina` is that the deterministic engine is called through abstract function pointers exposed by the OpenSSL `RAND_METHOD` API as shown in Listing 1.

This abstraction allows for the dynamic backend instantiation and replacement we have previously mentioned (yielding a very difficult static analysis).

Listing 1: OpenSSL `RAND_METHOD` API

```
struct rand_meth_st {
    int (*seed) (const void *buf, int num);
    int (*bytes) (unsigned char *buf, int num);
    void (*cleanup) (void);
    int (*add) (const void *buf, int num, double entropy);
    int (*pseudorand) (unsigned char *buf, int num);
    int (*status) (void);
};
typedef struct rand_meth_st RAND_METHOD;
```

NIST DRBGs DRBG stands for Deterministic Random Bit Generator, and they have been standardized by NIST with a functional model in their SP-800 90A publication [25]. This model is presented in Figure 10, and exhibits the four main functions of a DRBG:

— `Instantiate()`: this function takes a random seed as entropy input as well as optional nonce and personalization string, and initializes the DRBG internal state.
— `Generate()`: this function produces random bits, and takes as input optional additional data that add entropy to the internal state.
— `Reseed()`: this function is specifically called to add entropy to the internal state by adding dedicated entropy input as well as optional additional data.
— `Uninstantiate()`: internal state zeroization.

The critical data when it comes to the DRBG security are the entropy inputs during `Instantiate()` and `Reseed()`: they must remain secret for a safe DRBG usage (*i.e.* forward and backward secrecy of the generated random bytes). Although the additional data are not critical (*i.e.* the security does not rely on them), they add more non-determinism to the DRBG generation in an opportunistic way.

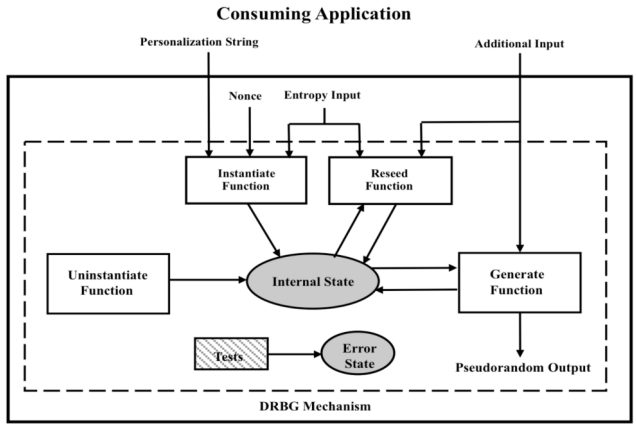

Fig. 10. NIST SP-800 90A DRBG functional model (source: the NIST standard)

Behind the functional aspect, there are three possible engines that compute the internal state and the inputs/outputs [7]:

— CTR-DRBG: based on AES or TDEA (Triple DES) block cipher in counter mode.

[7] One can refer to [30] for an exhaustive C implementation covering all the NIST DRBGs.

— Hash-DRBG: based on hash functions (the list of standardized ones are SHA-{1,224,256,384,512,512-224,512-256}).
— HMAC-DRBG: based on HMAC using the previous list of standardized hash functions.

These various NIST DRBGs have a history of security analysis [12, 34, 35], with CTR-DRBG being the most performant at the price of a less clean design compared to Hash-DRBG and HMAC-DRBG. None of these have practical attacks against them when properly instantiated and reseeded though. Explicit `Reseed()` is usually a good practice for internal state refreshing, but in practice all the implementations follow the standard that mandates an implicit reseed when many kilobytes of random have been generated (while explicit reseed is optional and not used). Finally, it is worth noticing that a fourth DRBG has been part of the standard: DUAL-EC-DRBG [23]. It has been withdrawn in 2014 following an alleged NSA trapdoor in the BSAFE library and a $10 million deal with RSA Labs [18].

In the case of ASA firmware, among the various random related engines, two of them are of interest: a CTR-DRBG using AES-256 and a Hash-DRBG using SHA-512 are used during ECDSA and RSA key generation.

MD_RAND Before DRBGs were standardized by NIST, almost every cryptographic library used its own pseudorandom generation custom method for post-processing entropy. The common ground was the usage of APIs that are similar to an Instantiate/Generate/Reseed where entropy is injected at initialization and then when reseeding, and optionaly when generating random. MD_RAND is OpenSSL (old) way of performing this: it is based on MD-5 or SHA-1,[8] uses an entropy pool and state of around one kilobyte. An interesting particularity of MD_RAND is that reseeding is explicit, and to limit a no-reseeding impact the content of the output buffers when performing a `Generate()` are systematically taken as additional input data to get opportunistic entropy.[9]

MD_RAND makes calls to the `RAND_poll()` function behind the scene at initialization, which gathers entropy for the initial pool and internal state. `RAND_poll()` is hence critical and is expected to provide

[8] Only the SHA-1 hash version is used in Cisco ASA firmwares.
[9] The interested reader can best visualize this behavior by looking at “Line 467” on slide 13 of [17], as this is the least critical of the two elements at the origin of the Debian “PURIFY” CVE-2008-0166 [16] where only the PID of the processes was used as an entropy source.

high quality entropy: it is usually plugged to an OS systemic entropy source such as `/dev/urandom` under UNIX systems or `CryptGenRandom()` under Windows (for recent versions of OpenSSL). In CiscoSSL, the Cisco ASA OpenSSL fork, we have discovered that although a `/dev/urandom` is present (we are on a Linux system), a proprietary implementation of `RAND_poll()` was used with bad sources, yielding weaknesses for MD_RAND.

4.3 Entropy sources and entropy lifting

Now that we have described how the DRBG and MD_RAND deterministic engines are used, it is obvious that the random bytes generation is as robust as the entropy injected at instantiation, reseeding and to a lesser extent during generation (with optional additional data or input buffers content for MD_RAND).

We distinguish here entropy sources that are raw values coming from collecting points, and entropy lifters that usually process these sources in multiple samples to extract one useful random entropy seed that can be provided as input to the upper layers (usually the deterministic engines). The idea behind lifters is to improve the number of entropy bits for low quality collecting points. The theory behind entropy sources, their quality measurement and their stochastic modeling is vast and is not the subject of this article. The curious reader can refer to NIST SP 800-90B [26] and BSI AIS20/31 [4] for comprehensive guidelines.

In this section, we will focus on the main entropy sources we have analyzed during ECDSA and RSA certificates keys generation, and as we will see most of them are bad. As usual, we emphasize the fact that we are not exhaustive in the enumeration of these sources: we only describe the ones that are used by the firmware we have analyzed on the platforms we took a look at.

`rand()` The standard library `rand()` non-cryptographic PRNG, based on a Linear Congruential Generator (LCG), has been seen directly used as one of the entropy sources by `RAND_poll()`. It is used without any previous call to `srand()`: this means that all the bytes produced by this source are obviously always predictable. The only small uncertainty is the sampling offset (by the deterministic engine) in the produced stream because `lina` multi-threading makes multiple consumers of `rand()` compete for this resource. From our tests, exhausting a small window of a few bytes is enough to cover all the possible runtime cases.

`gettimeofday()` Time is often a "cheap" entropy source when used correctly. A bad idea is to use predictable time values, such as boot time or absolute time in seconds as these are obviously guessable by an attacker depending on the context and information he has on the target platform. Using timings with higher volatility and noise is a better idea although not considered as a strong entropy source. For instance, POSIX `gettimeofday()` provides a time with microseconds resolution: on devices with multi-core and a high frequency CPU, sampling `gettimeofday()` in a program will produce values with variable low bits (high bits corresponding to seconds are evidently predictable).[10] Cisco makes use of `gettimeofday()` to feed their DRBGs with additional data. We have however discovered that two variants are used in the firmware versions we have analyzed. Some versions use the POSIX `gettimeofday()` provided by the standard library, but other versions use a **custom version** of this API completely implemented in software in `lina`: the elapsed time is measured from the `rdtsc` CPU cycles instruction and rounded to multiples of 10 milliseconds, removing a lot of volatility and hence entropy. This has disastrous consequences on the randomness generation as we will see in the next section.

`rdtsc` and LFSR or LCG extender As a variation of time measurement, CPU cycles measurement can be a "cheap" entropy source. The `rdtsc` instruction provides a 64-bit value representing the number of cycles elapsed on the CPU core since reset. On CPUs with high frequency (typical Intel or AMD ones), the upper bits are stable while the lower bits are more volatile: on a 2 GHz CPU, around 11 bits are flipped every microsecond. Hence one sample provides a few bits of entropy, and getting multiple samples to mix them together (e.g by concatenating and then hashing the result) is a good way of lifting this entropy. Unfortunately for Cisco, we have unveiled another method used in `lina` that lacks robustness: the 32-bit low part of an `rdtsc` sample is used as a seed in a Linear Feedback Shift Register to produce PRNG bytes. The chosen LFSR is in Galois mode, it uses a 32-bit primitive polynomial and has a maximum length period of $2^{32} - 1$ bits, but the whole system provides anyhow at most 32 bits of entropy!

From the analysis of the Cavium firmware, a Linear Congruential Generator (LCG) is used in place of the LFSR to lift input 32 bits of the

[10] This assertion is less true on devices with a very deterministic behaviour, which is generally the case for small microcontrollers or real-time systems.

CPU cycles (a dedicated Octeon register that is equivalent to `x86 rdtsc` is used). This lifting is as lousy as the LFSR based one.

Unitialized or initialized buffers, ASLR This "entropy source" was kind of unexpected when we discovered it (and it seems to be actually a happy coincidence in `lina`). During our analysis, we have observed that some MD_RAND input buffers (hence used to add entropy to the state) contribute to cryptographic material generation. Unitialized and initialized buffers with static data are kind of simple to handle as it is sufficient to observe them once to know them (this is easy on instrumented platforms, a lot more difficult on non-instrumented ones). Other input buffers will contain addresses (from the stack, from the heap), and coincidently add real entropy to MD_RAND thanks to ASLR. For the kernel versions we were dealing with in our firmwares, only 28-bit entropy are present in these addresses, which allowed a simple exhaustive search.

Cavium hardware backed entropy When the Cavium cryptographic accelerator is present, the deterministic engines try to use hardware backed entropy: the `lina` executable asks for randomness through PCI commands. Because we lacked instrumentation on the hardware platforms, we did not investigate much these aspects. A static analysis of the Cavium firmware showcases both good and bad entropy sources: while Cavium dedicated RNG IP is used to provide high quality random bits,[11] various fallback paths exist to lower quality sources such as the `rdtsc` with LCG combo. These elements are discussed further in Section 5.

Good sources in `lina` On the `x86` side, and depending on the firmware versions, some good quality sources can be exploited. `rdrand` and `rdseed` instructions with proper lifters are present in the binary and could be used on the CPUs that support them. One drawback though is that many Cisco ASA hardware platforms have Intel CPUs that lack this support (*e.g.* Intel Atom or AMD Geode), and for the ASAv virtual images it is complex to ensure that the hypervisor will expose such capabilities of the CPU (this will of course depend on the underlying physical CPU capabilities, but also on the hypervisor configuration and so on).

[11] The hardware RNG is based on ring oscillators [5].

5 Keygenning ASAv

This section details the work performed on ASAv firmwares to validate the understanding of the entropy sources and processing performed in various firmware versions to produce the RSA modulus, ECDSA private key and ECDSA signature nonces during Cisco ASA self-signed certificates generation at boot.

The section first provides some statistics on observable collisions from a black-box standpoint for a set of firmware versions. Then, the details of a keygen written for a specific version which does not exhibit collisions (9.10.1-44) are given. The rest of the section builds on this detailed description to elaborate on the way keygens for 5 more versions (9.6.4-36, 9.8.1, 9.8.2/9.9.1, 9.8.3) have been developed.

5.1 Statistics on ASAv

For the reasons detailed in section 3.3, instrumentation of ASAv takes time and we only focused on a limited number of firmware versions. Table 11 provides the blackbox statistics gathered (using our `qemu` automation script) for RSA and ECDSA certificates generated by those versions.

Firmware	RSA mod.	ECDSA `x`	ECDSA `r`	#generated
9.6.4-36				100
9.8.1				93
9.8.2	●	■		56
9.8.3	■	■	■	25
9.9.1	●	■		60
9.10.1-44				232
9.12.2-9				100
9.16.1				100

Fig. 11. Black box statistics on various ASAv firmware versions

In table 11, an empty box indicates no observable issue for the item on that specific version in our setup. A disk ● indicates duplications for the item on that specific version. A square ■ indicates duplications but only when boot time is the same, *i.e.* a boot time dependency for random generation. Disks or squares colored (● and ■) indicate collisions shared between versions.

The black squares ■ for version 9.8.3 for RSA modulus, ECDSA private key and ECDSA signature nonce indicate that the collisions only occur

when the system boot time is the same. This indicates that the boot time value participates in the entropy used for generation of those elements but this also means - considering collisions do occur - that there is not much more participating to the entropy of the system. This will be covered in Subsection 5.3.

The green disks for 9.8.2 and 9.9.1 indicate that both versions of firmware have collisions for RSA modulus, but also indicate that identical modulus values were shared between them, *i.e.* the entropy sources and processing code that provide the random number for RSA modulus generation is shared. Still for those versions, the green squares for ECDSA public keys indicate that collisions do occur for each version but only for matching boot time and that both versions do share values, *i.e.* the entropy source and processing code is identical for the generation of this item and include a dependency to boot time that does not exist for the RSA modulus generated sooner in the process. The last interesting details on those versions is that no ECDSA nonce duplication is visible from the certificates data. This will be covered in more details in Subsection 5.3.

From an external standpoint, the empty boxes in the table for other versions indicate no observable collisions. As we will see later in this section, this does not mean that those versions have decent entropy in the generation process of keys and nonces. This will be covered in Subsection 5.2 for version 9.10.1-44 and 5.3 for version 9.8.1.

5.2 Detailed example of ASAv 9.10.1-44

Summary Having a high level understanding of the mechanisms involved in the output of random values during the boot of an equipment is one thing. Being able to generate the set of random values that may be provided at each point during the boot is another story: it requires a precise understanding of **all** the aspects that are involved.

Instead of providing a complete description of each keygen developed for the ASAv firmware versions given previously, we decided to focus on the 9.10.1-44 version which was the most challenging and exhibited most of the interesting elements that we had to deal with during all the keygenning work we did.

To ease reader's understanding of this section, we first start with a short summary of how 9.10.1-44 produces random before going in the details of each step and mechanism.

From a high level perspective, the initialization of main random source on this version on our setup can be described in the following way (we will

denote the interesting steps that will be later analyzed using the circled ⓧ notation for step x):

— **In a first thread**:

1. CTR-DRBG initialization is requested. For this purpose, 40 bytes and 20 bytes of entropy are requested to the MD_RAND engine for entropy input and nonce.
2. MD_RAND not being initialized yet, the first request leads to its initialization, which requires 32 bytes of seed.
3. The request for those 32 bytes of seed to initialize MD_RAND are performed to the CTM layer.[12]
4. Among the possible providers that CTM may use on the platforms it supports, it ends up calling unseeded `rand()` in our case to provide those 32 bytes.

— Then, **in the main thread** :

1. CTR-DRBG is initialized with
   — 40 bytes and 20 bytes of entropy grabbed from MD_RAND for entropy input and nonce
   — A personalization string including current time from boot expressed in multiples of 10 ms (step ❶).

— As was done in first thread, MD_RAND is **reinitialized** using a 32 bytes random value. We will see that the fact that the MD_RAND initialization is performed in the first thread has an impact on the way this initialization goes.

— The 32 bytes used by MD_RAND for its initialization are taken from a LFSR lifter, which is called for the first time, resulting in a seeding from a 32 bits value provided by `rdtsc`.

Then, after the initialization of this CTR-DRBG in the main thread, which will serve as the default RNG for this thread, the usual set of calls to this default RNG are performed by all the successive functions called (see Figure 8) during the boot of virtualized Cisco ASA devices:

— A call requesting 16 bytes for the BSAFE RNG backend (step ❷)

— 4 sets of 3 calls each requesting 16 bytes for `SSL_CTX_new()` [13] (steps ❸ to ⓮).

— A call requesting 28 bytes for the RSA seed (FIPS 186-4 method B.3.4) for generating the RSA certificate (step ⓯).

[12] CTM is an abstraction layer developed by Cisco for various possible cryptographic helpers which depend on hardware, software and runtime combinations.

[13] Those 3 random values grabbed by each `SSL_CTX_new()` are used as keys to protect TLS session tickets.

— A call requesting 32 bytes for the ECDSA private key (step 16).
— A call requesting 8 bytes for BSAFE RNG backend, which then provides 32 bytes of BSAFE RNG random for ECDSA signature nonce during ECDSA certificate signature (step 17).

All the calls listed above are performed through the main OpenSSL `rand_bytes()` interface but result in calls to the `generate()` method from the CTR-DRBG initialized in the main thread. As we will see in more details, the callbacks providing additional data during each `generate()` call include current time from boot expressed in multiples of 10 ms.

At that point of the section, the attentive reader can already conclude that - even if multiple layers of RNG are stacked which will include states and do transformations steps - the real entropy available at each call is limited to:

— 32 bits of RDTSC.
— Uninitialized data found in 40 bytes and 20 bytes input buffers to be filled by MD_RAND for entropy input and nonce.
— Set of timing values.
  — Feeding the CTR-DRBG personalization string.
  — Feeding the CTR-DRBG additional data during generate() calls.

The details regarding those elements and the final articulations of the keygen will be covered in the next subsections. The details of the impacts of the initialization of the first thread on the second one will also be discussed in next subsections.

Listing 2: CTR-DRBG initialization C code

```c
drbg_ctx ctx;
int drbg_initialized = 0;

int CiscoSSL_DRBG60_init()
{
    struct timeval tv;
    unsigned char pers_buf[64];

    drbg_set_type(ctx, AES256_CTR);
    drbg_set_entropy_nonce_callbacks(drbg_ctx,
                                     drbg_get_entropy_nonce_cb);
    drbg_set_rand_callbacks(ctx,
                            drbg_get_adin_cb,
                            drbg_rand_seed_cb,
                            rand_add_cb);
    memcpy(pers_buf, "CiscoSSL DRBG60", 16);
    get_time(&tv);
    ((unsigned int *)pers_buf)[12] = tv.sec;
    ((unsigned int *)pers_buf)[13] = tv.usec;
    ((unsigned int *)pers_buf)[14] = 1;
    drbg_instantiate(ctx, pers_buf, 64);
    drbg_initialized = 1;
}
```

CTR-DRBG instantiation 9.10.1-44 version has the same kind of generic initialization code for CTR-DRBG as other versions using this mechanism. A simplified version of this initialization code can is represented in Listing 2.

The function starts by setting the DRBG flavour (AES256-CTR) and then sets the various callbacks that will be used during the operations of the DRBG:

- `drbg_get_entropy_nonce_cb`: this callback is called internally by the `drbg_instantiate()` function to first grab 40 bytes and then 20 bytes to be used respectively as entropy input and nonce input for DRBG instantiation. The subsystem to which the request is performed is MD_RAND, *i.e.* `drbg_get_entropy_nonce_cb()` is a simple wrapper around MD_RAND main random provider (`ssleay_rand_bytes()`). As it will be discussed below, this implies that MD_RAND is the main source of entropy used for DRBG instantiation on this ASAv firmware version in our setup.
- `drbg_get_adin_cb`: this callback is used to provide 48 bytes of additional random input at each DRBG `generate()` call to be used to refresh the DRBG state. Simply put, the implementation of this callback is a wrapper around the `get_time()` function which we will discuss in more details below. Let's spoil a bit by telling that even though the output is a 48 bytes buffer, the only "entropy" it contains is the output of `get_time()`.
- `drbg_rand_seed_cb`: this callback would be used to request entropy if a DRBG reseed was called at some point. It is never called in practice.
- `rand_add_cb`: this callback is the one that will be called when OpenSSL `rand_add()` RNG method is called by user code to voluntarily push new entropy and refresh the RNG state of current OpenSSL subsystem. As an example, when OpenSSL `BN_rand()` is called to generate a random big number integer, *e.g.* in the process of generating a new ECDSA key, `rand_add()` is explicitly called to pass current time value on 8 bytes. This should lead to a dependency of ECDSA key to boot time but - as will see below - this is not the case.

The remaining of the function comes and fills a 64 bytes buffer whose content is mainly fixed; the only varying part is provided by the `get_time()` function which returns a time value encoded on 8 bytes.

The nonce and entropy buffers are - with the personalization string discussed above - the random root seeds of CTR-DRBG instantiation.

```
Thread 2 hit Breakpoint 4 in ?? ()
$6 = "========= DRBG pers string is:"
$7 = 0x1
0x7fffea1a0bc0: 0x43 0x69 0x73 0x63 0x6f 0x53 0x53 0x4c     // CiscoSSL
0x7fffea1a0bc8: 0x20 0x44 0x52 0x42 0x47 0x36 0x30 0x00     //  DRBG60\0
0x7fffea1a0bd0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00     \
0x7fffea1a0bd8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00      \  Lot of 0
0x7fffea1a0be0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00      /
0x7fffea1a0be8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00     /
0x7fffea1a0bf0: 0x06 0x00 0x00 0x00 0xe0 0x04 0x07 0x00     // time
0x7fffea1a0bf8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00     // counter
```

Fig. 12. An example of `pers_buf` content grabbed using `gdb`

The only variability in the personalization string being the time value (more on this below), the CTR-DRBG instantiation fully depends on the quality of the random provided by MD_RAND at that point.

Unlike some other firmware versions which use `gettimeofday()` to collect time values since epoch with a precision to the microsecond the function we called `get_time()` returns the time elapsed in a `struct timeval` but the content of the `tv_usec` field contains a value which is only accurate to a multiple of 10ms. In the personalization buffer presented in Figure 12, the boot time is 6s and 460000us, which is indeed a multiple of 10ms. The 13 bits of higher entropy are lost in this puzzling rounding process resulting in at most 7 bits of entropy in the personalization string.

Then, because the 48 bytes of additional data returned by `drbg_get_adin_cb()` callback during `generate()` calls also use `get_time()`, the additional entropy brought during each call after the first one does not depend anymore on the intrinsic boot time value but only on the difference between the time included in a given step (*e.g.* the personalization string) and the following one (next call to `get_adin_cb()`).

Figure 13 provides 125 recorded timings in ms from boot on the Y axis for each of the 17 interesting first steps of CTR-DRBG operation after boot (numbered ❶ to ⓱ as previously defined on the X axis). We can observe:

— The low volatility in boot time ❶ (from 1910 ms to 3130 ms by 10ms increment, *i.e.* 7 bits).
— Little volatility for step ❷ and ❸ (up to 50ms, up to 30ms respectively).
— Near zero time spent in the 4×3 steps associated with the 4 `SSL_CTX_new()` cases, *i.e.* at most one bump of at most 10ms.

— Up to 20ms (1 bit) between steps ⓮ and ⓯.
— Up to 20ms (1 bit) until next ⓰ for RSA seed.
— Between 110ms and 350ms between RSA seed generation and ECDSA nonce generation: this large and a bit more volatile value (5 bits) is due to the varying time of the FIPS algorithm depending on RSA seed value.

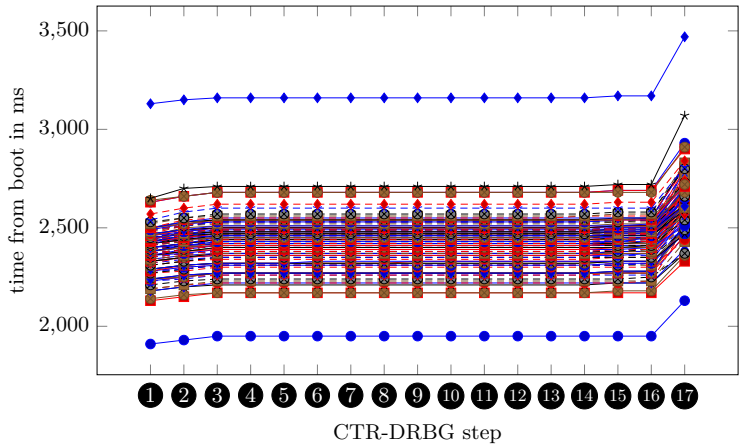

Fig. 13. Recorded timings for CTR-DRBG for steps ❶ to ⓱ (125 captures)

The little volatility is better seen on Figure 14 that also work on the same dataset of 125 runs but presenting only the delta in ms between previous CTR-DRBG steps. Note that the value for step ❶ has been set to 0 to avoid a squeezing effect on the plot; let us just keep in mind the 7 bits of time entropy for that specific step.

One can conclude that during the `generate()` calls in `SSL_CTX_new()` the delta between the outputs of `get_time()` in `get_adin_cb()` are almost all 0, as discussed above, with very few of them having a value of 10ms (in steps ❹ to ⓮ included). The 3 points above 0 for those steps are for different runs, confirming that at most a single toggle between a time value t and $t + 10$ms only occurs once between steps ❹ to ⓮ included. This observation help making the **brute-force** practical as will be seen later in this section. This would definitely not have been the case without the rounding performed by `get_time()`.

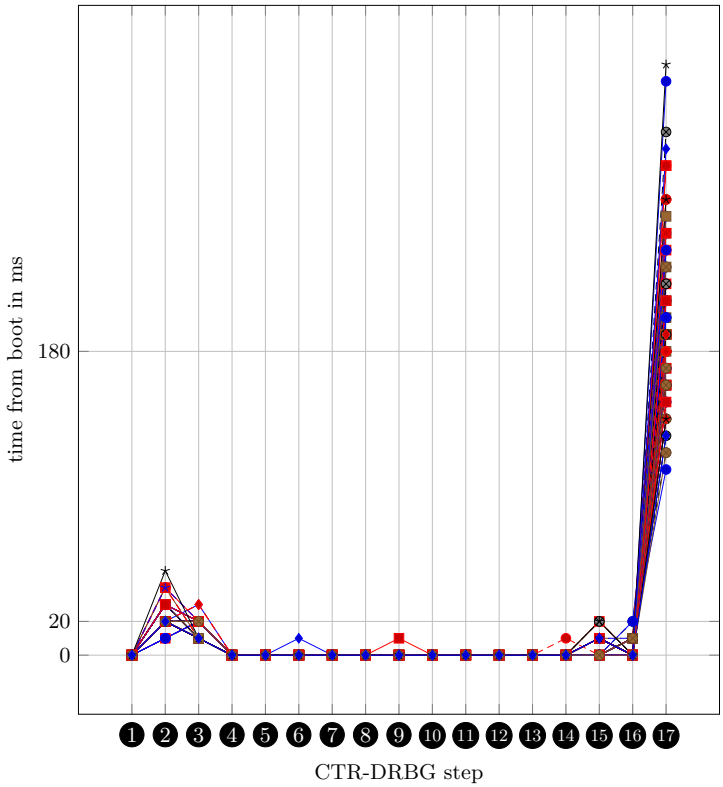

Fig. 14. Delta timings for CTR-DRBG (125 captures)

MD-RAND We saw that CTR-DRBG initialization depends on MD_RAND initialization to provide 40 bytes of entropy and 20 bytes of nonce, through calls to `ssleay_rand_bytes()`.

In practice, the situation is a bit more complex than expected because there is not a single MD_RAND initialization, but two, in 9.10.1-44 version as presented in the introduction of this subsection. For that reason, even if only the second thread participates in the generation of RSA and ECDSA certificates, the impacts of what happens in the first one need to be understood. The first initialization of CiscoSSL DRBG60 in the Thread #1 will happen using MD_RAND for the first time to provide entropy and nonce: 32 bytes are requested using a CTM function capable of selecting various entropy sources (hardware backed TRNG, `rdrand`, etc.). On our ASAv setup, the 32 bytes are extracted from calls to (unseeded) `rand()` whose output is fixed and known.

The second thread will also instantiate a CTR-DRBG, but with a semi-reinitialized MD_RAND context (as `ssleay_rand_cleanup()` has been called in between).[14] A major difference here is that MD_RAND's `RAND_poll()` has also switched its random source backend, and now uses `rdtsc` with the LFSR lifter.

Once this instantiation performed, the CTR-DRBG acts on its own, the only external dependency being the time added by `get_adin_cb()`.

Assembling things in a practical keygen Creating a keygen for this version requires a few additional steps to validate the composition of the mechanisms presented in previous subsections:

- Validation of the LFSR output (compared to other firmware versions).
- Validation of MD_RAND initialization and use of input buffers, including stability of these buffers in different setups (with and without `gdb`). On other versions (*e.g.* 9.8.2), some of MD_RAND input buffers have toggling bytes, or include varying addresses due to ASLR.
- Validation of CTR-DRBG work.
- Progressive setup of the two aspects of the bruteforce: `rdtsc` seeding followed by bruteforce of timing values used in CTR-DRBG operations (`instantiate()` and `generate()`).

To cut the problem in half and avoid launching a large bruteforce without guarantees of success, we first created a patched ASAv image with a tiny binary modification [15] providing a fixed seed during call to `rdtsc`. Rebuilding a `.qcow2` image with that lina binary and generating a set of certificates provides the expected result: a good amount of collisions on RSA and ECDSA certificates. Generated certificates are then used as a basis to validate the CTR-DRBG part of the keygen (*i.e.* the timing bruteforce). We followed the same patching strategy to force some values provided by `get_time()` and validate our CTR-DRBG with incremental keygenning.

The 2^{32} `rdtsc` seeds with the LFSR lifter can be transferred to parsing a 2^{32} bits keystream for a time-memory tradeoff (allowing to skip the seed derivation processing through precomputed tables of a few gigabytes). When we take all the possible timings for the CTR-DRBG rounded to 10ms, we have around 2^{13} possible tuples when limiting the delta timings

[14] We do not detail this here, but a remanent static variable in MD_RAND state is important in the generation process.

[15] See subsection 3.3 for the details of running modified images on ASAv.

to the largest plausible values (and limit their sum to the total possible running time). This leads to a complexity of around 2^{45} for the exhaustive search of keys with "heavy" DRBG computations at each step. Although this can be achieved in a few weeks on a HPC cluster or using costly cloud computation, we have decided to follow another path with a poor man's PoC that would work on a regular machine.

From the statistics of timings extracted from Figure 14, we have picked the most probable values for each step to drastically reduce the number of tuples and target a $\approx 2^{37.5}$ complexity. Using the patched `rdtsc` based binary, we have observed a rate of 1.7% certificates broken with the chosen timing tuples.[16] To validate our keygen, we expected to observe the same rate of broken certificates when generated using the unpatched firmware. And this is indeed what we obtained: over $1,000$ new certificates produced on the unpatched binary, a certificate of the set is broken every ≈ 9 hours on a 16 CPUs Intel Xeon machine (32 hyper-threaded cores). Finally, ≈ 8.5 days were necessary to complete the exhaustive search.

A relevant question about this keygen is its adherence to the underlying (physical or virtual) platform producing the certificates, *i.e.* does our specific setup somehow biases the certificates production? While there is a clear dependency on the CPU microarchitecture and *e.g.* the presence of `rdrand` instructions (we have used a virtualized Nehalem CPU without such instructions [17]), we have tried to check that we do not have more specificities. Regarding the 10ms rounded timings, most of the modern `x86` CPUs are fast enough to produce the same timing profiles: we have used three various Intel CPUs virtualizing ASAv and validated that the same keygen allows to break the certificates. Similarly, the number of CPUs dedicated to `kvm` does not seem to interfere with our keygenning (we have tried with 1 and 4 CPUs in the `kvm` configuration).

5.3 Other versions

For the sake of conciseness, we only presented a somewhat detailed explanation of the keygenning of the 9.10.1-44 version. Even if other versions also had some very interesting variations and challenges, we only cover them briefly to give the reader a glimpse of the various (unsuccessful) combinations of RNG and seeding method we encountered.

ASAv 9.6.4-36 In this version, RSA seed and ECDSA private key are both generated using CTR-DRBG as a random provider. The ECDSA

[16] This rate has been observed with a $1,000$ certificates black-box generation campaign.
[17] Mainly because we believe it captures the average machines in the wild.

nonce is generated from a completely different deterministic engine: the Hash-DRBG (which is in fact also indirectly used as an entropy provider for the CTR-DRBG).

This Hash-DRBG runs from the following (lack of) entropy parameters:

— An empty personalization string is used for instantiation.

— The entropy input and nonce used during instantiation are respectively 32 bytes and 16 bytes of a LFSR stream initialized with 32 bits of `rdtsc`.

— No additional inputs are used during generate calls.

The CTR-DRBG shares the same initialization pattern as for 9.10.1-44: entropy input and nonces are provided by MD_RAND. The only difference is that MD_RAND is seeded by the Hash-DRBG engine. This makes the CTR-DRBG harder to exploit since Hash-DRBG computations must be added and somehow break the 2^{32} time-memory tradeoff used for `rdtsc` with slower computations.

However, since the ECDSA nonces generation only relies on Hash-DRBG, we are able to keygen them in 2^{32}. Once this is done, recovering the ECDSA key is mathematically easy (getting the RSA seed from the ECDSA key is not so trivial as reversing the DRBG from the output should be intractable).

ASAv 9.8.1 In this version, RSA seed and ECDSA private key are both generated using CTR-DRBG and the ECDSA nonce is generated using the Hash-DRBG (this is somewhat similar to 9.6.4-36). The CTR-DRBG runs with similar parameters except that the entropy input and the nonce are extracted from unseeded `rand()`, and that real `gettimeofday()` with microseconds precision is used for `generate()` calls, rendering its keygenning unachievable in reasonable time: direct keygenning of RSA and ECDSA keys is not feasible. Nonetheless, as for 9.6.4-36, ECDSA nonces can be broken with a complexity of 2^{32}, yielding in ECDSA immediate key breaking.

ASAv 9.8.2/9.9.1 Version 9.8.2 and 9.9.1 do share the same RNG code basis and outputs. In those versions, MD_RAND is the main deterministic engine. RSA seed and ECDSA private key are generated directly from MD_RAND. ECDSA Nonce is generated from the BSAFE RNG seeded from MD_RAND.

MD_RAND is instantiated from 32 bytes taken from unseeded `rand()`, and the input/output buffers during `rand_bytes()` are filled with either fixed known values (zeroes or other data), almost fixed values (one toggling

byte), or with stack or heap addresses with ASLR. Hence, keygenning these versions bring some interesting challenges as we have to break ASLR,

The analysis of the dynamic behavior of this version and the validation of the understanding through the development of a keygen helps understand the black-box statistics presented in 5.1:

— The lack of dependency to time for RSA modulus generation results from the use of MD_RAND which - unlike Cisco initialization and use of DRBG - does not include the time as entropy value.
— The dependency to time for the ECDSA private key generation is explained by the fact that OpenSSL does perform a `rand_add()` of time before calling `rand_bytes()`, which creates an obvious dependency to time after that point for random extracted from MD_RAND.
— The lack of visible collisions for ECDSA `r` components of the signature results from the presence of ASLR during its generation (bringing a weird situation with lucky good randomness).

From our analysis, ASLR for our buffer on the studied firmwares has only 28 bits of entropy (due to the concerned addresses that are aligned on 8 bytes and the inherent limitations of ASLR in the concerned kernel). This allows for a keygen complexity of 2^{28} plus $\approx 2^5$ for the toggling bytes and index in the `rand()` buffer [18] (due to multi-threading), yielding a total complexity of $\approx 2^{33}$.

ASAv 9.8.3 In this version, CTR-DRBG is the main random engine. RSA seed and ECDSA private key are both generated directly from CTR-DRBG. The ECDSA nonce is generated from BSAFE RNG seeded from CTR-DRBG.

The only sources of entropy are unseeded `rand()` for the entropy input and nonce in `instantiate()`, the system boot time rounded to 10ms for the personalization string, and 10ms timings for additional inputs of `generate()`.

Keygenning this version boils down to exhausting all the possible tuples of boot time and delta timings rounded to 10ms while handling some `rand()` offsets toggling due to multi-threading, which represents around 2^{16} when we consider plausible timings limits. This is performed quickly on a regular laptop.

[18] The only time used here is immediately extracted from the certificate `notBefore`.

5.4 Summary

A summary of all the analysis and keygen work performed on ASAv is presented on Figure 15. The highlighted versions are proven vulnerable while not concerned by the previous CVE-2019-1715.

Firmware	RSA modulus	ECDSA nonce	ECDSA key	Comment	Keygen complexity
ASAv9.6.4-36	◗	●	◗ ▲	`CTR-DRBG` is seeded by `MD_RAND`, itself seeded by HASH-DRBG itself seeded by a LFSR itself seeded by `rdtsc` rounded to 32 bits	2^{32} (nonce)
ASAv9.8.1		●	▲	`CTR-DRBG` "saved" by addin with true `gettimeofday()`, `HASH-DRBG` seeded by a LFSR itself seeded by `rdtsc` rounded to 32 bits	2^{32} (nonce)
ASAv9.8.2	●	●	●	`MD_RAND` seeded by `rand()`, ASLR in input buffers for `MD_RAND` (nonce), BSAFE seeded by `MD_RAND`	$\approx 2^{33}$
ASAv9.8.3	●	●	●	`CTR-DRBG` seeded by `rand()` BSAFE seeded by `CTR_DRBG`	$\approx 2^{16}$
ASAv9.9.1	●	●	●	`MD_RAND` seeded by `rand()`, ASLR in input buffers for `MD_RAND` (nonce), BSAFE seeded by `MD_RAND`	$\approx 2^{33}$
ASAv9.10.1-44	❍	❍	❍	`CTR-DRBG` seeded by `MD_RAND` seeded by LFSR seeded by 32 bits `rdtsc`. Bad `gettimeofday` is also used.	Full: $\approx 2^{45}$ PoC: $\approx 2^{37.5}$

Legend:
● Fully broken with a PoC **keygen**
❍ Broken with a PoC **keygen** with higher time complexity
◗ **Fragile** entropy sources, harder to exploit (but seems feasible)
▲ Broken as a side effect of nonce breaking
Versions highlighted are vulnerable and NOT concerned by previous CVE-2019-1715

Fig. 15. ASAv firmwares keygenning overview

6 Investigating RNG failure on hardware devices

As previously stated, we did not have the opportunity to investigate instrumentation of hardware appliances. Instead, we have only performed a black-box analysis of the produced certificates. The current section first presents the results, then discusses our basic investigations on the Cavium firmware, and then concludes with some hypothesis on the root causes and possible future work.

6.1 Black box statistics on 5506-X

Table 16 presents the black box statistics we obtained on firmware versions ranging from 9.6.2 to 9.16.

Firmware	RSA modulus	ECDSA `r` nonce	ECDSA `x` key	#generated
9.6.2-23				45
9.6.3-20				15
9.6.4-34	❶		❷	15
9.6.4-36	❶		❷	15
9.6.4-40	❶		❷	15
9.6.4-41	❶		❷	15
9.6.4-42	❶		❷	15
9.6.4-45	❶		❷	45
9.7.1-4				160
9.8.1				60
9.8.2	❸	◗	|❹|	60
9.8.3		❺		60
9.8.4-10		❺		10
9.8.4-41		❺		30
9.9.1	❸	❺	|❹|	30
9.9.2-85		❺		30
9.10.1-44		◗		30
9.12.4				30
9.12.4-35				30
9.13.1-12				30
9.14.3-18				30
9.15.1-15				30
9.16.2-14				30
9.16.2				45

Legend:
● = collisions shared between firmware versions
◗ = isolated collisions
|●| = collisions emerging with same certificate time
Same number/color = collision values shared across versions
Empty box = no observable collisions, inconclusive
Versions highlighted are vulnerable and NOT concerned by CVE-2019-1715

Fig. 16. 5506-X black-box statistics

A few elements are worth commenting:

— There are visible RSA and ECDSA key or nonce duplications for a wide range of versions between 9.6.4-34 and 9.10.1-44. This means that although there is a hardware backed cryptographic accelerator providing physical random sources, either it is used but badly configured, or another software backend is used. In any case, this is the symptom of a very dubious behaviour.

— Some versions show shared values (trail of colors), which means that the same deterministic engines are used with the same entropy sources and inputs for the concerned duplicated values.
— Some versions (9.8.2, 9.9.1) exhibit collisions only when the boot time is set, implying the usage of time as an entropy source.
— When compared with table 16 for ASAv, we can spot interesting differences. 9.6.4 and 9.10.1-44 versions show collisions in 5506-X where nothing is exhibited on ASAv (in black-box at least). 9.8.3 suffers from only nonce collisions on 5506-X while RSA and ECDSA keys are also impacted on ASAv. The usage of fixed boot time exhibits new collisions but not at the sames places, which is the sign that parts of engines or entropy sources might be the same but other parts might differ.
— As for ASAv, empty boxes do not mean that there is no issue at all. As we have proven in the previous sections with advanced keygenning on the virtualized platforms, no visible collisions do not mean no entropy issue. This is why we have marked these cases as inconclusive.

From the pure results and differential diagnosis, we can speculate on some hypothesis. 9.8.2 and 9.9.1 share the same behaviour on both hardware and virtualized platforms, but this behaviour is different on each. The behaviour for ECDSA and RSA private keys are exactly the same as in ASAv: the MD_RAND backend seems to be used there, with certificate time addition when ECDSA key is generated.[19] On the other hand, ECDSA nonces exhibit a different behaviour since observable collisions exist on 5506-X and not on ASAv (where ASLR hides them), and these collisions do not depend on time. This is the sign that another deterministic backend and/or other sources are used (*e.g.* Hash-DRBG instead of MD_RAND). This might be a coincidence (or not), but these specific two firmware versions are among the ones explicitly concerned by the original CVE-2019-1715 [10]. The highlighted versions exhibit a vulnerable behaviour while not concerned by this previous CVE.

To summarize, we can conclude that ASAv and 5506-X seem to sometimes share the same code paths and backends (with apparently a failure of the Cavium based entropy sources on 5506-X), and sometimes not, yielding in firmware versions that might be more fragile on one type of platform or another, and with differences for specific key material (RSA modulus, ECDSA key and nonce).

[19] Unfortunately, because of too much unknown input buffers, we cannot confirm this with keygenning.

6.2 A quick tour of Cavium firmware

In the current section we try and explore the possible paths that could lead to the Cavium hardware backend failure as an entropy source for `lina` in the platforms using it. For this, we focus on the Cavium firmware analysis that bring some trails to follow. The SoC is embedding a main MIPS64 BE processor with additional specific Octeon instructions and hardware blocks accessed through dedicated registers and memory addresses.

The Cavium firmware is a 2MB stripped binary loaded by the main `x86` CPU at boot time on hardware platforms via the PCI interface. Cisco has used the Cavium SDK for many of the low-level functions, but the proprietary communication post-boot over PCI seems to have been specifically developed by Cisco on top of the SDK. The `lina` analysis also exhibits some binaries with a proprietary format (embedded in the executable) that seem to be sent to the Cavium firmware at runtime for TLS and IPsec.

Unfortunately, the Cavium SDK as well as emulators (that could have ease the dynamic instrumentation of the firmware) are only available under NDA. This is why we were only limited to static analysis using tools that support the specific MIPS N64 ISA and the Cavium specialized instructions.

We have discovered that third party libraries are included in the Cavium firmware, such as OpenSSL 0.9.7d (which is a quite old one). For all the 5506-X firmware versions we have studied (from 9.6 to 9.16), the Cavium SDK version and the used libraries seem to be stable with not much evolutions.

Although not completely satisfactory from an intellectual perspective (it is hard to draw solid conclusions), we have decided to list the findings that emerged from our static investigations and that we consider as bad practices that could lead to a flawed randomness generation. As a disclaimer, the elements presented hereafter must be taken with a grain of salt as no dynamic analysis confirmed them.

From the static analysis, Cisco firmware for Cavium has at least two main deterministic engines that may participate to the generation of random upon request from `lina`: an OpenSSL MD_RAND and a Hash-DRBG engine.

Looking respectively at the OpenSSL `RAND_bytes()` and Hash-DRBG `generate()` methods provides interesting leads. A common

entropy-providing source function [20] is called by both functions. The `hw_get_random()` function implementation shows two main paths: if the processor flags indicate availability of a hardware RNG (which seems to be the case for most recent Octeons), the function will use it. Else, a fallback will grab 64 bits of a `rdtsc`-like value and will pass the 32 lower bits to `LCG_init()` as a seed of the Linear Congruential Generator entropy lifter that operates modulo 2^{64} (the LCG used is the one described in [15] page 106), and random is then provided through calls to `LCG_get_random()` using the LCG. To sum up, if no hardware RNG is available on the Cavium or if this detection fails, each call to `hw_get_random()` to get *e.g.* 16 or 32 bytes of random will produce buffers biased output with only 32 bits entropy. Finally, `RAND_bytes()` also contains another entropy lifter that makes use of an AES-256 CBC and CTR PRF (that we could not relate to any standard we know [21]), making use of Octeon hardware accelerated AES instructions.

7 Conclusion

7.1 Bad random does not necessarily collide

One of the first conclusions of the work presented in this article is that even some versions initially thought not to have issues from an external standpoint ended up being keygened. This comes as a reminder that the quality of randomness cannot be assessed neither by looking externally at generated values in a limited set nor on a single platform.

The auditor may be capable of aligning boot time values, get enough samples to hit some birthday paradox but it is impossible to cover all the causes of bad random not colliding (*e.g.* inclusion of to-be-signed message as entropy input during signature, presence of a variable address in an input buffer, etc.).

Looking at external values is limited as it only deters superficial issues. The only ways forward are a validated (ideally simple and clean) design and validated entropy sources.

7.2 Mixing sources in a single RNG

During our journey, we witnessed stacks of random processing layers, each one used as a seeding source for the one above, with sometimes a

[20] We called it `hw_get_random()` but it may have another name in the original source code.

[21] But this can be static analysis bias.

unique low entropy source used by lowest layers. In the end, this complex stacking only contained almost the same amount of entropy than the one provided by the low entropy source.

As demonstrated in the section detailing keygens 5, the security of such a design and the resulting keys, nonces, etc. almost only depends on the secrecy of the design, *i.e.* on the ability for an (motivated) attacker to understand how the layers work together, directly violating Kerckhoffs's second principle.

Instead of spending time on stacking RNG layers, a more simple, logical and efficient approach is to select a state-of-the art RNG scheme and mix different entropy sources while keeping the following in mind:

— There is no point in having low entropy fallback mechanisms, *i.e.* one should consider an entropy source as acceptable if it can alone support the expected strength of the final mechanism. This avoids ending up calling `rand()` or using a LFSR seeded with 32 bits of `rdtsc` if nothing else is available.

— In random processing stacks return values of functions should be checked conscientiously. Failure MUST not result in the use of returned buffers, but in a final error or a set of retries possibly leading to a final error. As a practical example, now ubiquitously used `rdrand` and `rdseed` instructions return an error code; upon error, those instructions **guarantee** that the (expected random) return value will be null (as in "equal to 0 on all its bits"). This is only a single example of the possible impact of not checking error codes.

— In some versions of Cisco products, the design for sources selection is simply based on availability, *i.e.* the first available source is selected. Considering the importance of random sources for the whole security of the product, this approach can be improved by mixing multiple sources. This simple defense-in-depth advice will help to further reduce the impact of a single failure, without adding too much burden on the design.

7.3 The second best friend of a DRBG is a good `addin` method

The first best friend being a good `Instantiate()` method. As written in DRBG specifications,[22] the mechanism is deterministic in its operations and only depends on entropy input sources:

— the one used during `Instantiate()` (entropy and nonce),

[22] the 'D' as the beginning of DRBG serves as a hint.

— the one used during `Generate()` calls to provide additional inputs,
— the one used during `Reseed()` (if it ever happens).

Our work confirms that failure to base DRBG operation on a robust entropy source for initial entropy during `Instantiate()` makes initial random output guessable to an attacker. It also confirms that failure to provide decent additional inputs during `Generate()` (low entropy values or correlated values) makes the situation persist in later calls. It should be added that decent additional inputs are not a solution to a bad initial entropy source.

From both design and validation standpoints, the main focus during DRBG integration should be to guarantee that the following hypothesis hold:

1. Initial entropy during `Instantiate()` MUST come from a strong entropy source or a combination of strong entropy sources.
2. Additional input provided at each `Generate()` call SHOULD provide additional strong entropy.
3. `Reseed()` MUST be called with fresh entropy as often as possible.

7.4 A word on using time in random subsytems

During our analysis, we stumbled upon various uses of time-related values as entropy sources: performance counters, `rdtsc`, absolute time, rounded versions, etc.

Stating the obvious, using a low entropy absolute time value as a random source is pointless. Using a high precision (micro or nano seconds) value can provide a few bits of entropy but is definitely not a decent or sufficient entropy source for seeding a system wide mechanism like a DRBG. On a system which expects to perform cryptographic tasks (read: most system deployed nowadays) and hence which requires a decent entropy source for its post-processing mechanism, no developer should start playing with time primitives to extract bits of entropy, expecting an happy end to the story. The system (hardware or virtual appliances) should be designed to include serious entropy sources (TPM, dedicated chips, `rdseed`, etc.) available for random subsystem.

One could argue that there is no reason not to use `rdtsc` (for instance) **as a defense in depth mechanism**, to participate to the additional inputs of a DRBG. Although this is true, the point here is that defense-in-depth must be backing a sound design based on strong sources.

7.5 Horizontal and vertical impacts

One lesson learned while working on this study is that failing at providing correct random can have two kinds of impacts:

— Vertical: it can impact all the cryptographic aspects of a product, from certificates, to (EC)DH values for session keys, nonces, cookies, tokens, or TLS ticket protection keys.

— Horizontal: the problem with random related vulnerabilities is that bad cryptographic material may have been generated that needs quite some time to be replaced. This is what happened to Debian [16]. Regarding the current work, it is unclear when (and how) the amount of online Cisco ASA devices with vulnerable certificates will come to 0.

7.6 State of the art

Designing and implementing a good random subsystem on a platform is not an easy task but is definitely feasible, considering the amount of helpers now available at different levels. The main way to reduce the possibility of mistakes on this path is to understand the state of the art: both by working with standards that define useful rules [1, 4, 26] and by learning from others' mistakes to avoid repeating them. We hope that articles and CVEs exposing poor entropy issues and their disastrous consequences will be a wake-up call for developers and security architects to comply with these advices.

References

1. ANSSI. Référentiel Général de Sécurité, version 2.0, Annexe B1.
2. ANSSI. CVE-2023-20107. `https://www.cert.ssi.gouv.fr/avis/CERTFR-2023-AVI-0266/`, 2023.
3. Arnaud Ebalard. x509-parser: a RTE-free X.509 parser. `https://github.com/ANSSI-FR/x509-parser`.
4. BSI. A proposal for: Functionality classes for random number generators. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Certification/Interpretations/AIS_31_Functionality_classes_for_random_number_generators_e.html`, 2020.
5. Cavium Networks. Random number generator (United States Patent 6954770). https://www.freepatentsonline.com/6954770.html, 2005.
6. Cisco. Cisco Adaptive Security Appliance Software and Firepower Threat Defense Software Low-Entropy Keys Vulnerability. `https://sec.cloudapps.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-asa5500x-entropy-6v9bHVYP`, 2023.

7. EVE-NG. EVE - The Emulated Virtual Environment For Network, Security and DevOps Professionals. https://www.eve-ng.net/.
8. failoverflow. Console Hacking 2010, PS3 Epic Fail. 2010.
9. GNS3. The software that empowers network professionals. https://www.gns3.com/.
10. Greg Zaverucha. CVE-2019-1715: Cisco Adaptive Security Appliance Software and Firepower Threat Defense Software Low-Entropy Keys Vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-asa-ftd-entropy, 2019.
11. Heninger, Nadia and Durumeric, Zakir and Wustrow, Eric and Halderman, J. Alex. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security Symposium*, pages 205–220. USENIX Association, 2012.
12. Viet Tung Hoang and Yaobin Shen. Security analysis of NIST CTR-DRBG. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 218–247. Springer, 2020.
13. Jacob Baines. BlackHat USA 2022: Do Not Trust the ASA, Trojans! https://i.blackhat.com/USA-22/Thursday/US-22-Baines-Do-Not-Trust-The-ASA-Trojans.pdf, 2022.
14. Jatin Kataria, Rick Housley, Joseph Pantoga, and Ang Cui. Defeating cisco trust anchor: A case-study of recent advancements in direct fpga bitstream manipulation. In *Proceedings of the 13th USENIX Conference on Offensive Technologies*, WOOT'19, page 5, USA, 2019. USENIX Association.
15. Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1997.
16. Luciano Bello. CVE-2008-0166: DSA-1571-1 OpenSSL – predictable random number generator . https://www.debian.org/security/2008/dsa-1571, 2008.
17. Luciano Bello and Maximiliano Bertacchini. Predictable PRNG In The Vulnerable Debian OpenSSL Package - The What And The How. https://fahrplan.events.ccc.de/congress/2008/Fahrplan/attachments/1245_openssl-debian-broken-PRNG, 2008.
18. Matthew Green. Hopefully the last post I'll ever write on Dual EC DRBG. https://blog.cryptographyengineering.com/2015/01/14/hopefully-last-post-ill-ever-write-on/, 2015.
19. Mohamed Traore. Analyse des biais de RNG pour les mécanismes cryptographiques et applications industrielles. https://www.theses.fr/2022GRALM013, 2022.
20. NCC group. Cisco ASA blog series. https://research.nccgroup.com/2017/09/20/cisco-asa-series-part-one-intro-to-the-cisco-asa/, 2017.
21. NCC group. Cisco ASA-related projects. https://github.com/nccgroup/asatools, 2017.
22. Nils Schneider. Recovering Bitcoin private keys using weak signatures from the blockchain. 2013.
23. NIST. Recommendation for Random Number Generation Using Deterministic Random Bit Generators (old version with Dual-EC). https://csrc.nist.gov/publications/detail/sp/800-90a/archive/2012-01-23, 2012.

24. NIST. Digital Signature Standard (DSS). https://csrc.nist.gov/publications/detail/fips/186/4/final, 2013.
25. NIST. SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final, 2015.
26. NIST. SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation. https://csrc.nist.gov/publications/detail/sp/800-90b/final, 2018.
27. QEMU. QEMU TCG Plugins. https://qemu.readthedocs.io/en/latest/devel/tcg-plugins.html.
28. Rapid7 Labs. Rapid7 Labs Open Data. https://opendata.rapid7.com/.
29. RSA Labs. What are BSAFE and JSAFE? http://security.nknu.edu.tw/crypto/faq/html/5-2-3.html.
30. Ryad Benadjila, Arnaud Ebalard. libdrbg: a library implementing NIST SP800-90A DRBGs. https://github.com/ANSSI-FR/libdrbg.
31. Ryad Benadjila, Arnaud Ebalard, Jean-Pierre Flori. libecc: a library for elliptic curves based cryptography (ECC). https://github.com/ANSSI-FR/libecc.
32. Shodan search engine. Search Engine for the Internet of Everything. https://www.shodan.io/.
33. Chao Sun, Thomas Espitau, Mehdi Tibouchi, and Masayuki Abe. Guessing Bits: Improved Lattice Attacks on (EC)DSA with Nonce Leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022, Issue 1:391–413, 2022.
34. Joanne Woodage and Dan Shumow. An analysis of NIST SP 800-90a. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 151–180. Springer, 2019.
35. Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls hmac-drbg. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2007–2020, New York, NY, USA, 2017. Association for Computing Machinery.