Practical state recovery attacks against legacy RNG implementations

Shaanan N. Cohney
University of Pennsylvania
shaanan@cohney.info

Matthew D. Green
Johns Hopkins University
mgreen@cs.jhu.edu

Nadia Heninger
University of Pennsylvania
nadiah@cis.upenn.edu

ABSTRACT

The ANSI X9.17/X9.31 pseudorandom number generator design was first standardized in 1985, with variants incorporated into numerous cryptographic standards over the next three decades. The design uses timestamps together with a statically keyed block cipher to produce pseudo-random output. It has been known since 1998 that the key must remain secret in order for the output to be secure. However, neither the FIPS 140-2 standardization process nor NIST's later descriptions of the algorithm specified any process for key generation.

We performed a systematic study of publicly available FIPS 140-2 certifications for hundreds of products that implemented the ANSI X9.31 random number generator, and found twelve whose certification documents use of static, hard-coded keys in source code, leaving the implementation vulnerable to an attacker who can learn this key from the source code or binary. In order to demonstrate the practicality of such an attack, we develop a full passive decryption attack against FortiGate VPN gateway products using FortiOS v4 that recovers the private key in seconds. We measure the prevalence of this vulnerability on the visible Internet using active scans, and demonstrate state recovery and full private key recovery in the wild. Our work highlights the extent to which the validation and certification process has failed to provide even modest security guarantees.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; **Embedded systems security**; **Security protocols**;

ACM Reference Format:

Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. 2018. Practical state recovery attacks against legacy RNG implementations. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243756

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243756

1 INTRODUCTION

Random number generation is a vital component of any cryptographic system. While systems may survive subtle flaws in cryptographic algorithm implementation, the ability to predict the output of a (pseudo)random number generator typically leads to the catastrophic failure of any protocol built on top of it. In recent years a number of cryptographic systems have been found to include flawed random and pseudorandom number generation subsystems. These flaws range from subtle weaknesses, *e.g.* biases that admit sophisticated attacks against the protocol [48]; to catastrophic vulnerabilities that allow for adversarial recovery of all random coins used in a protocol execution [16, 58]. In a particularly ominous development, some of these flaws appear to have been deliberately engineered. For example, leaks by Edward Snowden indicate that the NIST Dual EC DRBG standard may have been designed with a backdoor [52]. While there is no way to empirically verify this allegation, we know for certain that the Dual EC algorithm *has* been successfully exploited: in 2015 Juniper Networks revealed that their ScreenOS line of VPN devices had been modified to include a malicious set of Dual EC parameters, which likely enabled passive decryption of VPN sessions [16].

The problem of constructing random and pseudorandom number generators has been extensively explored by industry [8, 40, 47] and in the academic literature [20, 21, 41, 54, 56]. Despite the abundant results of this effort, the industry has consistently relied on a small number of common pseudorandom number generation algorithms. To a large extent this can be attributed to standards bodies. For example, until 2007 there were only *two* algorithms for pseudorandom number generation approved for U.S. FIPS 140 certification, and prior to 1998 only one such algorithm was approved. Recent discoveries surrounding the inclusion of flawed generators motivate a more thorough examination of these generators — and particularly, their use in products.

The ANSI X9.17/31 standards. The ANSI X9.17 "Financial Institution Key Management (Wholesale)" standard, first published in 1985, defined a voluntary interoperability standard for cryptographic key generation and distribution for the financial industry. This standard included a pseudorandom number generator (PRG) in its Appendix C as a suggested method to generate key material. This generator uses a block cipher (in the original description, DES) to produce output from the current state, and to update the state using the current time.

The same PRG design appeared in US government cryptographic standards for the next three decades, occasionally updated with new block ciphers. A subset of the ANSI X9.17-1985 standard was adopted as a FIPS standard, FIPS-171, in 1992. FIPS-171 specified that "only NIST-approved key generation algorithms (e.g., the technique defined in Appendix C of ANSI X9.17) shall be used". FIPS 140-1, adopted in 1994, specified that modules should use a FIPS approved key generation algorithm; FIPS 186-1, the original version of the DSA standard adopted in 1998, lists the X9.17 PRG as an approved method to generate private keys. The ANSI X9.31 standard from 1998 specified a variant of the X9.17 PRG using two-key 3DES as the block cipher; this variant was included as an approved random number generator in further standards such as FIPS 186-2, from 2004. NIST published extensions of this design using three-key 3DES and AES as the block cipher [39] that were officially included on the FIPS 140-2 list of approved random number generation algorithms in 2005.

A critical design element of the ANSI X9.17/X9.31 PRG is that the cipher key used with the block cipher remains fixed through each iteration. In order to remain secure, the key must never be revealed to external attackers. If an attacker learns this key, they can recover *all future and past states* of the PRG from its output by brute forcing the timestamps [41]. Perhaps due to this known weakness, the ANSI X9.17/X9.31 design was deprecated in 2011 and removed from the FIPS list of approved PRG designs in January 2016. NIST SP 800-131A, the document announcing the deprecation of this algorithm, also deprecated a number of smaller cryptographic key sizes along with a rationale for doing so; no rationale appears to have been given for the transition away from X9.31.

Despite this significant flaw, which was identified by Kelsey *et al.* in 1998 [41], the NIST documents specifying the ANSI X9.31 PRG design fail to specify how the cipher key should be generated [39]. This raises the possibility that even FIPS-validated deployed systems could contain vulnerabilities that admit practical PRG state recovery. To evaluate this possibility, we performed a systematic study of publicly available FIPS 140-2 certification for hundreds of products that implemented the ANSI X9.31 random number generator.

Our results show that a number of vendors use static hard-coded keys in source code, leaving them vulnerable to an attacker who can learn this key from the source code or binary. In order to demonstrate the practicality of this attack, we reverse-engineered the binaries for a Fortigate VPN gateway using FortiOS version 4. We discovered that the ANSI X9.31 PRG implementation used for IPsec encryption uses a hard-coded key, which is a test value given in the NIST RNGVS specification [43], published as a validation suite alongside their standardization of the generator. We perform full state recovery in under a second from random number generator output. We observe that a passive adversary observing the IKEv2 handshake used to set up an IPsec connection can carry out a state recovery attack using the plaintext nonce values in the handshake, and then derive the secret key generated during the cryptographic key exchange. We demonstrate a full attack that learns the session keys for a Fortigate IPsec VPN using FortiOS version 4 in seconds. Furthermore, we demonstrate that this vulnerability exists *in the wild* by performing state recovery, key recovery, and decryption on handshakes we collected using internet-wide scanning of VPN hosts.

This is not a "NOBUS" backdoor: it is symmetric, and thus an attacker with access to the source code or device can recover the secrets needed to compromise the PRG. However, the failure mode of static, discoverable keys we exploit was not ruled out by standards. The PRG appears to have been independently implemented in this fashion by a variety of vendors. This is a failure of the standardization process that has led to real and ongoing vulnerabilities.

We note that this failure mode is more subtle than simply using a hard-coded key for encrypted communications. There are many cryptographically secure PRG constructions using only public parameters and entropy unknown to the attacker, for example constructions based on modular exponentiation [11] or hash functions [8]. In addition, as we discuss in Section 7, the vulnerability could have been rendered practically unexploitable by using sufficiently high resolution timestamps.

1.1 Our Contributions

In this work we extend a growing line of research into weakened/backdoored random number generation in deployed products [9, 15, 16, 18, 19, 55]. Specifically, we demonstrate the existence of widespread and *passively* exploitable vulnerabilities in implementations of one of the most widely-deployed standard PRGs. Our contributions are as follows:

- We perform a systematic study of FIPS 140 security policy documentation and discover several independently vulnerable PRG implementations from different vendors, and discover critical failures in the standardization process.
- Based on this work, we develop an efficient passive X9.31 state recovery attack for the FortiOS v4 IPsec implementation and demonstrate full IPsec VPN decryption. Our reverse engineering was aided by a privilege escalation vulnerability we discovered and disclosed.
- We use Internet-wide measurements to measure the scope of this vulnerability among publicly-visible hosts, and demonstrate it against hosts in the wild, uncovering more than 25,000 vulnerable hosts. These numbers likely represent a small fraction of the true number of vulnerable hosts.
- We discuss the impact of these findings on other standardized PRG designs, and demonstrate that these vulnerabilities could affect other symmetric PRG implementations as well.

A critical differentiator between this work and previous work is that our work is the first to exploit flaws in a widely-used *symmetric-key* PRG at large scale, rather than a specialized (and rare) public-key design such as Dual_EC_DRBG [16]. Moreover, we note that our findings are tied to our analysis approach: they would not have been detectable through black-box external testing methods.[1] To our knowledge, this work is the first to identify exploitable flaws in cryptographic devices by analyzing the output of cryptographic

[1]Unlike many previous PRG weaknesses, *e.g.*, [34, 58] the PRG flaws in this work are undetectable to an attacker who interacts with the device as a black box; they can only be found through careful analysis of the PRG internals.

module validation procedures. This demonstrates that the existing standards validation procedures may need to be revisited.

Purpose of this work. Over the past several years, a valuable line of research has considered the impact of weakened number generators on cryptographic devices. This research comprises three categories of work: (1) discovery of novel cryptographic attacks [41, 55], (2) measurement and impact studies of known (theoretical) algorithm flaws [15, 16] and (3) development of countermeasures and new theoretical models [9, 18, 19].

We stress that these categories are mutually interdependent. Without knowledge of flaws, there can be no analysis of impact. Without knowledge of practical impact (in deployed protocols and devices), there is little impetus for theoretical analysis or countermeasure development. Finally, without academic research in each of these areas, it is difficult for industry and standards bodies to design or motivate analysis of new algorithms.

This work is an example of category (2). Our goal is to evaluate the *impact* of a specific flaw on real, deployed cryptographic systems. Our results in this work demonstrate that these flaws are present and exploitable at scale in widely used implementations of real Internet protocols. Our findings do not flow inevitably from previous results on ANSI X9.31 [41], as we note above and in Section 7. Moreover, this analysis is critical given that X9.31 is one of the most widely-deployed standard PRG algorithms in existence.

Finally, this work has impact in motivating countermeasure research and formal modeling. In particular, we note in Section 7 that more recent designs (CTR_DRBG from SP800-90A) may also be vulnerable to similar attacks if implementations include (minor) flaws. Since CTR_DRBG is enormously popular – e.g., it is included in every Intel processor – we believe our work motivates further analysis of implementations, as well as consideration of symmetric PRG design principles that improve robustness.

1.2 Disclosure

We disclosed the X9.31 and privilege escalation vulnerabilities to Fortinet in October 2016. Fortinet responded by releasing a patch for affected versions of FortiOS [24, 25]. FortiOS version 5 did not implement the X9.31 PRG and is not vulnerable.

We disclosed the potential for a flaw in Cisco Aironet devices to Cisco in June 2017. After an internal investigation, Cisco determined that the affected software versions had all reached end-of-support status. They were unable to find the source code to validate the flaw.

We notified the remaining vendors listed in Table 2 in October 2017. BeCrypt pointed us to version 3.0 of their library, which has been FIPS certified and no longer includes the X9.31 random number generator. They told us that the only fixed key inside the FIPS module is for self-test purposes. ViaSat USA had no record of the product indicated in the security documentation and ViaSat UK failed to respond to our disclosure. We did not receive substantive responses from any other vendors.

NIST has decertified the ANSI X9.31 PRG for FIPS compliant uses independently of our work. Despite this, we detected many vulnerable devices active on the open Internet, and additional devices may reside within enterprise networks. In personal communication in response to our work, NIST noted several issues with cryptographic validation that they are planning to improve; we discuss these in detail in Section 7. NIST informed us that they introduced a five-year sunsetting policy for FIPS 140-2 validations in 2016 in order to weed out old validations and encourage upgrades. There is an effort currently underway to transition to automated testing for all modules, and to change the liability model so that vendors carry full responsibility for the security of their products [49].

1.3 Ethics

While we demonstrate key recovery and decryption against live hosts we do not own on the Internet, the traffic we decrypt in our proof-of-concept is a handshake we initiated with this host. We did not collect traffic or attempt decryption for connections in which we were not a party. We followed community best practices for network scans, including limiting scan rates, respecting hosts who wished to be blacklisted, and working with vendors and end users to minimize effects on their networks.

2 BACKGROUND

2.1 Pseudorandom generators

We adopt the notation of Dodis et al. [19].

Definition 1 (Pseudorandom generator). A pseudorandom generator (PRG) is a pair of algorithms (I, G). The seeding algorithm $\mathsf{I}(\lambda)$ takes a security parameter λ and probabilistically generates an initial state $s \in \mathcal{S}$, typically some fixed-length bit string. The generation algorithm $\mathsf{G} : n \times \mathcal{S} \to \{0,1\}^n \times \mathcal{S}$ maps the current state to an n-bit output and a new state. For any λ, integer $q \geq 1$, initial seed $s_0 \in \mathsf{I}(\lambda)$, and any list of non-negative integers $(n_1, n_2, \ldots, n_q)$ we let $\mathrm{out}^{\mathsf{q}}(\mathsf{G}, s_0)$ denote the set of bit strings $(r_1, r_2, \ldots, r_q)$ produced by computing $(r_i, s_i) \leftarrow \mathsf{G}(n_i, s_{i-1})$ for $i = 1$ to q. A PRG is secure when no adversary can distinguish between the outputs $\mathrm{out}^{\mathsf{q}}$ and a set of random bits.

The PRG discussed in this work extends this basic definition slightly, as the generate function G also takes (and may return) *additional input*, namely a counter or timer value that is used as a partial input to the generator. We require that pseudorandomness hold even when this auxiliary data is predictable or adversarially chosen.

2.2 ANSI X9.31

The ANSI X9.31 random number generator is an algorithm that was included in some form on the list of approved random number generators for FIPS and NIST standards between 1992 and 2016. The design first appeared in the ANSI X9.17 standard on cryptography for the financial industry, published in 1985, using DES for the block cipher. The X9.31 variant uses two-key 3DES for the block cipher, and NIST published three-key 3DES and AES versions in

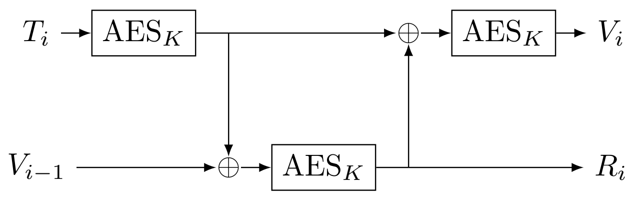

Figure 1: Each iteration of the ANSI X9.31 PRG generation function (G) inputs a timestamp T_i and a seed V_{i-1} and produces an output block R_i and a new seed V_i.

2005. [39] While this design has appeared under various names, we will refer to it as the X9.31 PRG for the rest of this paper, to use the terminology in modern implementations and standards.

The PRG is based on a block cipher with block size ℓ bits. We will specialize to AES, and define $\ell = 128$. $E_K(Y)$ represents the encipherment of Y under key K.

The seeding algorithm I selects an initial seed $s = (K, V)$ where V is generated randomly and K is a pre-generated fixed key K for the block cipher. The exact language used to describe the key in the NIST specification [39] for the AES-based variant is "For AES 128-bit key, let *K be a 128 bit key." and similarly for 192 and 256 bits. It continues "This *K is reserved only for the generation of pseudo-random numbers."

The j^{th} call to the generate algorithm G takes as input a desired output length in bits n, the current state $s = (K, V)$ and a series of timestamps $(T_1, \ldots, T_N)$ where $N = \lceil n/\ell \rceil$. Let $V_0 = V$ at the start of the generate call. For $i = 1$ to N the state is updated using the current timestamp T_i as follows. First, generate an intermediate value $I_i = E_K(T_i)$. Then one block of output is generated as

$$R_i = E_K(I_i \oplus V_{i-1}) \tag{1}$$

and the state for the next iteration is $V_i = E_K(R_i \oplus I_i)$.

The output of G is $\mathsf{truncate}_n(R_1 \| R_2 \| \ldots \| R_b)$ where $\mathsf{truncate}_n$ outputs the leftmost n bits, as well as the updated state $s' = (K, V_b)$. A diagram of the generation algorithm appears in Figure 1.

2.3 State Recovery Attack with a Known Key

We are not aware of a formal proof showing that ANSI X9.31 is pseudorandom, though this is likely to be the case if the block cipher is a pseudorandom permutation.

Kelsey et al. [41] observed that the generator is clearly vulnerable when K is not secret. An attacker who learns K can recover the current state using two consecutive blocks of output together with guesses for their timestamps. (A single block of output will not uniquely identify the state, but two blocks almost surely will.) Let R_0 be a block of output generated at T_0, R_1 a block of output generated at T_1, and $D(Y)$ the decryption of Y using key K. We can relate these quantities as:

$$D(D(R_1) \oplus E(T_1)) = R_0 \oplus E(T_0) \tag{2}$$

If the timestamps are only known approximately, we can brute force them within some range until we find a pair that yields equality, or apply a meet-in-the-middle attack [41]. If one block is not known completely, we can rearrange the encryptions and decryptions and verify equality of the known portion of the block. Once the timestamps T_0 and T_1 are known, the next seed is

$$V_2 = E(R_1 \oplus E(T_1))$$

A guess for the output from the next iteration is then uniquely defined by a guess for the timestamp T_2:

$$R_2 = E(E(T_2) \oplus V_2) \tag{3}$$

The above attack allows an attacker who has access to raw X9.31 output and the key K to recover the state. The attacker can then predict future output by running the generation algorithm with a guess for each subsequent timestamp. Alternatively, she can recover *previous* output blocks by "winding the generator backwards" and guessing earlier timestamps. Both attacks require the same effort.

In order to understand the impact on real cryptographic usage, we will describe how this attack works in theory in the context of popular cryptographic protocols.

2.4 Attacking X9.31 in TLS

Checkoway *et al.* [16] performed an in-depth analysis of the vulnerability of the TLS protocol to a compromised random number generator in the context of the Dual EC DRBG. The attack surface is similar for a vulnerable X9.31 implementation, with two key differences. First, the Dual EC backdoor is asymmetric, and thus only a party who generates the curve points used with Dual EC can detect the presence of the backdoor or exploit it, while the X9.31 vulnerability is symmetric, and any implementation that stores a fixed secret key is vulnerable to passive exploitation by an attacker who can recover this key. Second, the Dual EC attack requires at least 28 bytes of contiguous PRG output for an efficient attack, while the X9.31 attack can be conducted with fewer bytes.[2] This second restriction plays a major role in the cost of an attack on a protocol such as TLS or IPsec.

2.4.1 TLS Background. A TLS 1.0, 1.1, or 1.2 handshake begins with a client hello message containing a 32-byte random nonce and a list of supported cipher suites. The server hello message contains a 32-byte random nonce, the server's choice of cipher suite, and the server's certificate with a long-term public key. The server and client then negotiate shared secret keying material using the chosen asymmetric cipher. For RSA, the client encrypts a secret to the server's public key; for (elliptic curve) Diffie-Hellman, the server and client exchange key exchange messages. The client and server then derive symmetric keys from the negotiated shared secret

[2]In practice, given $(256-n)$ bits of contiguous generator output, Dual EC state recovery involves a guessing phase consisting of 2^n elliptic curve operations. This becomes costly for values of $n \geq 32$. By contrast, the ANSI attack requires only 128 bits of contiguous generator output for initial state recovery and a small portion of a second block to test for correctness. Given $(256-n)$ total bits the probability of recovering the wrong state is generally small ($\approx M * 2^{-(128-n)}$ when brute forcing over a timestamp space of size M) even when n is large.

and nonces, authenticate the handshake, and switch to symmetric encryption.

2.4.2 State and key recovery in TLS. If the X9.31 PRG is used to generate both the random nonce and the cryptographic secrets used for the key exchange, then an attacker could use the raw PRG output in the nonce to carry out the state recovery attack, and then use knowledge of the state to derive the secret keys. The 256-bit client or server random is exactly two blocks of AES output. Some TLS implementations include a 32-bit timestamp in the first 4 bytes of the nonce; in this case the attacker would have fewer than two full blocks, but the attacker will likely still recover a unique state. For a Diffie-Hellman key exchange, this attack would work if either the client or server uses the vulnerable PRG; for RSA key exchange, the key exchange would only be compromised if the client uses the vulnerable PRG.

2.5 Attacking X9.31 in IPsec

Checkoway *et al.* [16] describe the impact of a compromised random number generator on the IKE key exchange used in IPsec in the context of the Dual EC PRG. Our case is similar. We describe the protocols in detail, since we target IPsec for our proof-of-concept.

2.5.1 IPSec/IKEv2 background. IPSec is a Layer-3 protocol suite for end-to-end IP packet encryption, authentication and access control, widely used for Virtual Private Networks (VPNs). The IKE (Internet Key Exchange) protocols allow two hosts, denoted the Initiator and Responder, to establish an authenticated "Security Association", a secure communication channel. Two versions of IKE exist, IKEv1 and IKEv2. Both use Diffie-Hellman key exchange.

IKEv1. The original IKE specification [31] defines two phases, an initial key exchange phase (Phase 1) and a second phase (Phase 2) that uses keying material from the first phase to establish an IPSec SA. In Phase 1, authenticated key exchange can be performed using two handshake types: Main Mode or Aggressive Mode.

We focus our attention on the Phase 1 handshake in main mode. First, initiator and responder exchange Security Association (SA) payloads, with the initiator offering proposals for combinations of cipher suites and parameters and the responder accepting one. The parties then exchange Key Exchange (KE) messages, each containing a Diffie-Hellman key exchange payload. The format differs based on the authentication method. When using digital signatures or a pre-shared key to authenticate, the initiator and responder send their key exchange message together with a cleartext nonce of length between 8 and 256 bytes [31]. Each packet includes an 8-byte connection identifier called a cookie.[3] The ISAKMP specification (RFC 2408) [17] suggests that the cookie be generated by applying the MD5 hash function to the participant IPs, ports, and a local random secret.

Both parties then derive symmetric key material from the Diffie-Hellman shared secret, the nonces, the cookies, and optionally the PSK if using PSK authentication. All messages following this

[3]The ISAKMP specification (RFC 2408) [17] suggests generating the cookie by applying the MD5 hash function to the participant IPs, ports, and a local random secret.

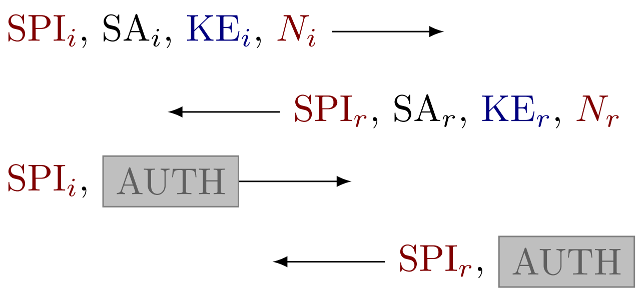
SPI_i, SA_i, KE_i, N_i
SPI_r, SA_r, KE_r, N_r
SPI_i, AUTH
SPI_r, AUTH

Figure 2: Randomness and the IKEv2 Handshake. The IKEv2 handshake establishes an authenticated, encrypted connection using a Diffie-Hellman key exchange. In our target implementation, both the SPI and nonce N are raw, unencrypted outputs from the PRG. The key exchange message KE is generated from the PRG immediately afterward. The encrypted portions of the handshake are inside of a gray box.

point are encrypted with the newly derived keys. Both sides then exchange certificates and identities, and authenticates the key exchange using the negotiated authentication method.

In Aggressive Mode, the initiator sends the SA and KE payloads together and the responder replies with its SA, KE and authentication messages together. IKEv1 Aggressive mode using pre-shared key authentication is widely considered to be a security risk because the authentication hash is sent unencrypted, which could allow an attacker to brute force the PSK.

In Phase 2, participants can negotiate additional keying material and exchange parameters using another Diffie-Hellman exchange, with messages encrypted using the key established in Phase 1. After negotiating this further material, the parties can exchange encrypted data.

IKEv2. The IKEv2 protocol was standardized in 2005 [14]. We show an abbreviated version of the IKEv2 handshake in Figure 2. First the initiator sends an IKE_SA_INIT message, with proposals similar to IKEv1, including a Diffie-Hellman public key generated using its best guess for the proposal parameters that will be accepted by the responder. Every message includes a connection identifier called the SPI. [4]

If the responder accepts the initiator's proposal, it replies with its own IKE_SA_INIT messages containing its key exchange. The two parties then authenticate each other and create an IPsec SA using IKE_AUTH messages, which are encrypted and integrity-protected using keys derived from the Diffie-Hellman shared secret, the nonces, and the SPI values. The analogue of Phase 2 in IKEv2 is the encrypted CREATE_CHILD_SA exchange, which admits an optional a second key exchange.

[4]In IKEv2 the cookie field from IKEv1 is renamed to the 'Security Parameter Index' (SPI). This is not to be confused with the IPSec SPI that identifies a particular SA, nor the IKEv2 COOKIE SA payload, which is a countermeasure against resource exhaustion attacks. The latter is called the IPSec SPI in IKEv1.

2.5.2 State recovery in IPsec. An attack on the IKE handshake exploiting a vulnerable X9.31 implementation proceeds much as described in [16]. The attacker requires that both the victim's nonce and Diffie-Hellman key exchange secret be composed of raw X9.31 output, and additionally, that the nonce be longer than one block in length. In an ideal attack scenario the Diffie-Hellman secret and nonce are generated in quick succession. The attacker then recovers the PRG state by guessing the timestamps used to generate the nonce, and checking for equality in Equation 2. The attacker then guesses the two timestamps used for the next two blocks of output using Equation 3, and confirms her guess using the public Diffie-Hellman exchange.

Full symmetric key recovery for IKEv1 depends on the authentication method used in the exchange. The attacker can validate state recovery and Diffie-Hellman secret compromise against a single key exchange packet from one side of the connection, but for some authentication methods may need additional information to generate the session keys. For signature authentication, the attacker does not need to learn any information beyond the nonces and cookies that appear in the clear in the handshake. For PSK authentication, the attacker would need to learn the PSK. For public key encryption authentication, the nonces are encrypted, so the attacker would need to learn the private keys for both sides of the connection in order to learn the nonces and derive the session keys.

For IKEv2, the IKE_SA_INIT messages contain all of the fields necessary to perform state recovery and derive the Diffie-Hellman secret: timestamps, nonces, the SPI nonce, and both key exchange values. We note that in IKEv2, the PSK is used only for authentication, and not to derive encryption keys. A passive attacker would need to collect both sides of the handshake in order to derive the session keys necessary to decrypt content, but state recovery and Diffie-Hellman secret compromise can be validated against a single packet from the vulnerable side of the connection.

3 FIPS AND HARDCODED X9.31 KEYS

As discussed in Section 2.2, the NIST design description for the X9.31 random number generator [39] does not specify how the block cipher key should be generated or stored. However, vendors who wish to obtain FIPS certification are required to produce a detailed public "security policy" document describing their cryptographic implementations and key management procedures. We performed a systematic study of the security policies for products certified for the X9.31 PRG to understand how many vendors publicly documented a potential hard-coded key vulnerability. We obtained the list of certified devices from the NIST web site [51].

3.1 Background on FIPS certification

FIPS 140-2 [50] defines requirements for cryptographic devices and software. This standard is used by the Cryptographic Module Validation Program (CMVP) to certify products used in US government applications. Compliant devices are eligible for certification under the CMVP jointly administered by NIST and the Communications Security Establishment (CSE) of Canada.

Certificate Type	2006-2008	2008-2016	To date
SP 800-90	0	1073	2053
X9.31/FIPS 186-2	310	952	1411

Table 1: Certificate issuances for X9.31 continued even after the publication of SP800-90 in 2006. The first SP800-90 CMVP certifications were issued in 2008, yet 47% of FIPS certificates issued 2008-2016 were for X9.31.

Once a device has been certified under the CMVP, it is added to a list of approved devices that US federal agencies and other regulated bodies may use.

FIPS 140-2 Annex C: Approved Random Number Generators listed the ANSI X9.31 Random Number Generator with AES and three-key 3DES between January 31, 2005 and the most recent revision on January 4, 2016; variants of the X9.17/X9.31 PRG using different block ciphers have been listed as approved random number generators in FIPS and NIST standards since at least 1992. In January 2011, NIST deprecated the X9.31 PRG in a transition away from smaller key lengths and weaker cryptographic algorithms [7]. Currently, the only approved PRGs are from NIST SP 800-90A, which was updated in June 2015 to remove Dual EC DRBG.

3.2 Certified unsafe usage of the X9.31 PRG

We examined the security policy documents of all devices certified under the CMVP that documented previous or current use of the X9.31 PRG. NIST provides a historical list of implementations certified for random number generators [1]. A single FIPS validation certificate may cover multiple products and versions. The scope of these certificates varied: in some cases they validated a cryptographic module or a single product and version, and in others they covered entire product lines and operating systems. According to this list, FIPS has issued 2,516 certificates in total for products that implemented X9.31. Of these, on July 13, 2017, 997 listed current support for X9.31 despite its official deprecation in January 2016. The remaining certificates were only available in updated versions that had removed details of historical X9.31 implementations. Of the 997 that indicated support for X9.31, 682 certificates from 288 vendors were validated for random number generation.

The security policy documents each contain a list of Critical Security Parameters (CSPs), which includes access control, key and parameter generation, and zeroization policies. We also looked for discussion elsewhere in the documentation of seed key generation. 127 of the vendors did not mention the AES key in the list of CSPs or elsewhere in the documentation. Since we are unable to determine whether the key was generated securely, we exclude these from further study. This left 161 vendors who mention seed key generation in some capacity.

We counted an X9.31 implementation as secure if the documentation stated that the key and the seed were user-generated, the output of another random number generator, contained any discussion of specifying sufficient entropy for the seed key, or a strategy

Vendor	Product Line	Language Used
BeCrypt Ltd.	BeCrypt Cryptographic Library	“Compiled into binary”
Cisco Systems Inc	Aironet	“statically stored in the code”
Deltacrypt Technologies Inc	DeltaCrypt FIPS Module	“Hard Coded’
Fortinet Inc	FortiOS v4	“generated external to the module”
MRV Communications	LX-4000T/LX-8020S	“Stored in flash”
Neoscale Systems Inc	CryptoStor	“Static key, Stored in the firmware”
Neopost Technologies	Postal Security Devices	“Entered in factory (in tamper protected memory)”
Renesas Technology America	AE57C1	“With the exception of DHSK and the RNG seed, all CSPs are loaded at factory.”
TechGuard Security	PoliWall-CCF	“Generation: NA/Static”
Tendyron Corporation	OnKey193	“Embedded in FLASH”
ViaSat Inc	FlagStone Core	“Injected During Manufacture”
Vocera Communications Inc.	Vocera Cryptographic Module	“Hard-coded in the module”

Table 2: FIPS 140-2 Security Policies Documenting Potential X9.31 State Recovery Vulnerabilities. Since the X9.31 RNG was removed from FIPS 140-2 in January 2016, many vendors have published software updates to remove X9.31 and updated their security policies accordingly.

to generate keys uniquely per device or per boot. In the case of a user-generated key, the onus would fall on the user to ensure that the key is securely generated and rotated as necessary. We did not study these cases further. The largest class of devices we evaluated as safe, generated the AES key on boot by seeding from a non-FIPS approved random number generator, most commonly the Linux random number generator. As an example of language indicating what we considered to be safe X9.31 key generation, the InZero Gateway security policy states that the “PRNG is seeded from `/dev/urandom`...this provides the PRNG with 256 bits of entropy for the seed key” [2]. The text includes additional commentary on the risk involved in using a weak random number generator for the purpose of FIPS validation. While `urandom` has had known vulnerabilities stemming from failure to properly seed on first boot of some classes of devices [34], we considered such usage safe for the purposes of this analysis. As another example, the 2012 FIPS 140-2 security policy for the Juniper SSG 140 [37], which was certified for the X9.31 generator, states that for the “PRNG Seed and Seed Key” “Initial generation via entropy gathered from a variety of internal sources.” There were 149 certificates (93% of the 161) in this class.

We counted an implementation as potentially vulnerable to a state recovery attack if the documentation stated that a single key was used for the lifetime of a device, particularly in cases where an external attacker would be able to learn this key. Unsafe devices had documentation indicating that the AES key was stored statically in the firmware or flash memory and loaded at runtime into the PRG. There were 12 vendors in this class, covering 40 product lines. We list these products together with the language used to describe seed key generation in Table 2.

3.3 Device-specific analysis

We were only able to gain access to the binary image for one of the products we identified as potentially vulnerable, a Fortinet operating system. We give more details on our investigation in the next section.

Cisco confirmed to us that X9.31 was used in Aironet 12.4-based branches for access points, Wireless Service Modules (WiSMs) and 4400 controllers using version 7.0. They were unable to locate the source code or confirm use of a hardcoded key, although they agreed with our interpretation of the certification language. They informed us that the 4400 controllers reached end of support in 2016, the WiSM modules reached end of support in 2017, and the 12.4-based branch of Cisco IOS software that supported X9.31 reached end of support at an unknown date. Another family of access points used the 15.3 branch of IOS, which uses NIST 800-90 and not X9.31. Cisco informed us that they no longer ship products using X9.31.

The BeCrypt Cryptographic Library Version 2.0 documentation states that the “RNG seed key” is “pre-loaded during the manufacturing process” and stored as “compiled in the binary”. Version 3.0 of the BeCrypt library no longer includes the X9.31 PRG. BeCrypt stated to us that “Except in one case when we use the RNG key creation routine we do not recycle the strong entropy output from one usage to be the input to the next usage. Instead, we use fresh entropy. In the one case where we recycle the strong entropy input,

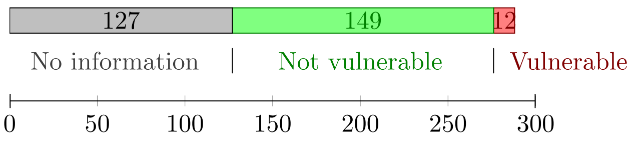

Figure 3: Counting vulnerable implementations. We examined the security policy documents from 288 vendors who had been FIPS 140-2 certified for the X9.31 PRG for information on how the seed key for the random number generator was generated. 12 vendors, or 4% of the total, documented a hard-coded key vulnerability.

Table 3: Affected Implementation Versions

Product Line	Version	X9.31 Removed
BeCrypt Cryptographic Library	2.0	3.0
Aironet	7.2.115.2	v8.0
DeltaCrypt FIPS Module	N/A	
FortiOS v4	4.3.17	4.3.18
LX-4000T/LX-8020S	v5.3.8	v5.3.9
CryptoStor	2.6	
Postal Security Devices	v28.0	v30.0
AE57C1	v2.1012	
PoliWall-CCF	v2.02.3101	
OnKey193	v122.102	
FlagStone Core	v2.0.5.5	
Vocera Cryptographic Module	v1.0	v2.0

the weak entropy input is actually strong entropy and the key is generated programmatically at startup" and additionally that the fixed RNG key inside the FIPS module is for self-test purposes.

The ViaSat's FlagStone Core documentation states that the key was "injected during manufacture". The documentation does not specify whether this key is device specific, although it recommends that "RNG Keys and Seeds that are imported into the FlagStone Core are generated or established using a FIPS 140-2 approved or a FIPS 140-2 allowed method." A device-specific key would require a targeted attack.

The certification documentation for Neopost devices specifies that the hardcoded key is entered in the factory and stored in tamper proof memory. A device-specific hardcoded key stored in tamper-proof memory would be quite difficult to attack.

3.4 Open source implementations

We also examined the X9.31 implementations in OpenSSL and the Linux kernel, but did not find evidence of hard-coded keys other than for testing.

4 DECRYPTING VPN TRAFFIC ON FORTIOS V4.3

The FIPS certification for FortiOS 4.3 states that the X9.31 key is "generated external to the module". We reverse engineered two versions of FortiOS and found that they used the same hard-coded key for their X9.31 implementation, which was then used as the system-wide random number generator.

We demonstrate that knowledge of this key allows an attacker to passively decrypt IPsec traffic from FortiOS v4. A PRG state recovery attack is feasible using only the IKE or TLS handshake nonces, and typically takes less than a second of computation time on our hardware, after which the attacker is able to guess the secret keys used to generate encryption keys. We performed an Internet-wide scan for affected hosts, and were able to carry out state recovery and private key recovery on handshakes from our scan data.

4.1 History of FortiOS 4.x

FortiOS is a network operating system created by Fortinet Inc. for their network security hardware devices and virtual appliances. In 2016, Fortinet was the fourth largest vendor by market share [3]. Fortigate primarily specializes in firewalls, intrusion detection systems and VPN gateways. FortiOS is used widely across their product suite.

FortiOS 4.0, released on February 20, 2009, included the X9.31 PRG. It was also included in the final major FortiOS v4 version, 4.3, released on March 18, 2011. It was not included in FortiOS 5, released in November 2012. Prior to our disclosure of the PRG vulnerability in October 2016, the last release of FortiOS v4 was 4.3.18, released August 6, 2014, with an end of support date of March 19, 2014 for devices compatible with FortiOS v5. In response to our disclosure of the random number generation vulnerability [5], Fortigate released version 4.3.19 of FortiOS in November 2016.

4.1.1 Vulnerabilities in FortiOS. On January 15, 2016, the MITRE corporation posted CVE-2016-1909 [4] revealing the presence of a hardcoded passphrase present in FortiOS 4.1.x and FortiOS 5.x as of October 2009, and all subsequent releases. This passphrase gave a remote attacker SSH access to the Fortimanager_Access account for remote administration. In a blog post in January 2016 [26], Fortinet stated that "This was not a 'backdoor' vulnerability issue but rather a management authentication issue... After careful analysis and investigation, we were able to verify this issue was not due to any malicious activity by any party, internal or external" and that the vulnerability had been patched in July 2014.

In August of 2016, a group calling themselves "The Shadow Brokers" released a collection of malware tools and documentation purportedly from an actor they termed"The Equation Group". Among other things, the leak contained a remote code execution exploit for FortiOS v3 and v4 titled EGREGIOUSBLUNDER. The exploit included code to identify FortiOS versions using HTTP response headers. The collection also included a malware payload for FortiOS (codename BLATSTING), containing a module 'tadaqueous' that disables random number generation by hooking the function get_random_bytes, the entry point to FortiOS's X9.31 implementation [57]. We did not find any evidence in the Shadow Brokers leak that The Equation Group was aware of the vulnerability we found in the PRG.

4.2 Static Analysis

We analyzed two implementations of FortiOS v4, the embedded operating system for Fortigate's network devices. The first was a firmware dump from a FortiGate 100D Firewall, and the second was a 'virtual appliance' (VM) running a different build of the operating system. The two firmware images were nearly identical, with minor variations due to the lack of hardware in the virtual appliance, and minor variations in supported TLS cipher suites. These differences would not have affected the measurements described in Section 5.

FortiOS is a GNU/Linux variant, with a customized shell that has kernel modules implementing hardware interfaces and cryptographic functions. The kernel is Linux 2.4.37, the last release of the 2.4.x series released in December 2008, which reached end of life in December 2011. FortiOS v5 still uses the Linux 2.4.37 kernel.

4.3 The X9.31 Implementation

The X9.31 random number generator is implemented within a kernel module that exports a Linux character device. At boot time, the init process loads the module and replaces `/dev/urandom` with a filesystem node corresponding to the X9.31 character device.

We reverse engineered the kernel module providing the X9.31 implementation and found the hard-coded AES key used for the PRG. (See Appendix A for the reverse engineered code.) The same key was used in both the firmware dump and virtual appliance. Although the documentation stated that the key was "generated external to the module", the key is the same one used for the NIST test vectors [?].

The PRG implementation generates timestamps using a call to do_gettimeofday() and produces a struct timeval containing the 64-bit time to the nearest microsecond. This struct is copied twice into a buffer to form a full 128-bit timestamp for the X9.31 generator.

4.4 The HTTPS Implementation

We also reverse engineered the implementations of the HTTPS server for the administration panel and the IKE/IKEv2 daemon used for VPNs. FortiOS v4 uses OpenSSL for TLS. When initializing the library, it sets the random number generation method to the system PRG, which is the X9.31 implementation.

The TLS server hello random consists of a four-byte timestamp followed by two raw blocks of X9.31 PRG output truncated to 28 bytes, which permits a state recovery attack. However, the TLS implementation does not seem to be vulnerable to a straightforward key recovery attack for Diffie-Hellman cipher suites via the server random because it uses ephemeral-static Diffie-Hellman. The secret exponent is generated when the server is launched and reused until shut down. In the case of RSA cipher suites, the client generates the encrypted pre-master secret for each session. The PRG vulnerability on the server does however affect initial RSA key generation.

4.4.1 RSA Key Generation. FortiOS generates the RSA keys used in its TLS certificates using OpenSSL's FIPS compliant routines calling the system X9.31 PRG for randomness. The primes it generates conform to FIPS 186, Appendix B.3.6, "Generation of probable primes with conditions based on auxiliary probable primes" [23]. For a 1024-bit modulus, each 512-bit prime factor p is generated using additional primes p_1 and p_2 so that $p_1|(p-1)$ and $p_2|(p+1)|$. This is intended to protect against Pollard's $p-1$ and Williams's $p+1$ factoring algorithms. This means that the primes that are generated have the form $p = r_p + p_0$ where p_0 is a 202-bit value derived from p_2 and p_1, and r_p, p_1, and p_2 are raw outputs from the PRG.

We generated a certificate on our VM and verified that the most significant bits of the RSA factors are related by Equation 2. However, this does not seem to lead to an feasible state recovery attack from the public modulus, since the attack requires raw PRG outputs. The only other call to the PRG during certificate generation produces a 4-byte serial number, insufficient for state recovery.

4.5 The IKE Implementation

The IKE daemon appears to be a modified variant of the raccoon2 project, compiled with the GNU MP library. All randomness used by the daemon is obtained by reading from `/dev/urandom`, and thus uses the X9.31 module. We analyzed both the IKEv1 and IKEv2 implementations to see if any fields in the handshake packets contained enough raw PRG output to permit state recovery.

In the IKEv1 implementation, the first block of PRG output is used to generate the IKEv1 cookie by hashing it together with IP addresses, ports, memory addresses, and the time since epoch, in seconds [5].

In the IKEv2 implementation, the SPI field, the equivalent of the IKEv1 cookie, is eight raw bytes of PRG output. In both IKEv1 and IKEv2, the next block of PRG output was used to generate the handshake nonce, which was 16 bytes long. This was generated immediately before the PRG output blocks that are fed into the Diffie-Hellman exponentiation.

For the case of Diffie-Hellman key exchange with the 1024-bit Oakley Group 2 prime, FortiOS v4 generates an exponent using two consecutive blocks from the PRG. In the virtual appliance's implementation, random bytes are read directly into the Diffie-Hellman exponent without modification. In the case of hardware devices with a dedicated cryptographic processor, the raw bytes of PRG output are fed along with the prime and the generator into a system call that invokes the cryptographic processor. This processor deterministically transforms the exponent in a way we were unable to reverse engineer, and outputs the result of the modular exponentiation.

We were able to invoke this system call ourselves on our hardware device to generate the Diffie-Hellman public key exchange values and shared secrets from candidate PRG blocks.

4.6 State recovery in IKEv1

The state recovery attack outlined in Section 2.3 requires two blocks of PRG output and the AES key to recover the state. The IKEv1 implementation gives us one full block of output in the nonce, and one block that is hashed together with a timestamp and nondeterministic pointers to create the cookie. The timestamp has a resolution of

[5]The IKEv1 cookie was `SHA1(0x2020||mpz_d||src||dst||timestamp||nonce`$_{16}$`)`. We note the choice of SHA1 here over MD5 recommended in the RFC. Here mpz_d represents a pointer to the buffer used by the linked gmp implementation that stores the remainder of the data to be hashed. This appears to be a quirk of using gmp types to store data, and not an intentional security measure on the part of the system implementer. The address itself is heap allocated, and was inconsistent across connections and restarts. The timestamp is seconds since epoch.

a second, so we assume it is known. However, the heap-allocated pointer provides approximately 13 bits of entropy [35]. Rearranging Equation 2, the first block of PRG output R_0 that is fed into the hash function to produce the cookie is $D(D(R_1) \oplus E(T_1)) \oplus E(T_0)$ where the second block of PRG output R_1 is known, and we estimate we need to brute force 29 bits of timestamps T_0 and T_1, as described in the next section. Thus an IKEv1 state recovery attack based on the cookie would take around 2^{42} hashes; which is feasible. However, we found that IKEv2 state recovery was cheaper, and focused our efforts on IKEv2 as described below.

The two blocks after the cookie and nonce are used to generate the Diffie-Hellman private key, which ensures that following state recovery, key recovery is straightforward.

4.7 State recovery in IKEv2

As discussed in Section 2.5.2, in order to recover the PRG state, we need two consecutive blocks of the PRG output, an approximation to the two timestamps used for the intermediate vectors, and the AES key. The FortiOS IKEv2 implementation yields 1.5 consecutive blocks of raw PRG output in the IKEv2 handshake: half a block in the SPI field, and a full block in the nonce. We learned the static key as described above by reverse-engineering the source code. We use the capture time of an incoming handshake to approximate the timestamps. From these, we use the approach described in Section 2.3 to recover the PRG state. We found that searching within a one-second window, or about 2^{21} guesses for the first timestamp, worked well on our hardware as well as scanned machines in the wild. [6]

We used our instrumented logging system to measure the time difference between successive calls to the PRG for the SPI and nonce fields that we needed to carry out state recovery on the FortiGate 100D. We found an average difference of $145\mu s$ with a standard deviation of $3.52\mu s$.

Using a 1 second window for the first block (2^{21} guesses), and bounding the search space for the second timestamp to within 3σ of our observed mean (2^5 guesses), yields a total state recovery complexity of about $2^{21} \cdot 2^5 = 2^{26}$ timestamp guesses.

Experimentally, this can be completed for either the virtual appliance or our hardware appliance in under one second on 12 cores of an Intel Xeon E5-2699 with parameters as above. For an expanded search space of 100 microseconds for the second timestamp as described in Section 5, successful runs completed in an average of 15 core minutes. Although we are verifying against only half a block of raw output from the first block, the reduced size of our timestamp search means that the expected number of false positive matches in Equation 2 is small, with probability at most $2^{26}/2^{64} = 2^{-38}$.

[6]The difference between the receive time of the first handshake packet received and the timestamp generated by `gettimeofday()` in the PRG when it was called to generate the SPI cookie is dependent on the time to execute the remainder of the packet creation and sending routine after the call to generate the timestamp, the time taken along the network to reach the attacking machine, the time taken by the attacking machine to process and report the packet, and clock drift.

4.8 State recovery in TLS

State recovery for TLS uses the 28 random bytes of the server random as 1.75 consecutive raw output blocks from the PRG. The first four bytes of the server random are a timestamp that help us fix a starting point for our search. Although we are verifying Equation 2 with 1.75 blocks of raw output, the reduced size of the timestamp search results means that false positives are very unlikely. For a timestamp search space of 2^{26}, we expect a false positive with probability $2^{26}/2^{96} = 2^{-70}$.

4.9 Recovering the IKEv2 Keys

Once we have recovered the PRG state from the SPI (block R_0 of output) and nonce (block R_1 of output), we can then wind forward the PRG, successively guessing the two following timestamps and applying Equation 1 to recover two more blocks R_2 and R_3 that will be used to generate the Diffie-Hellman secret.

We calculate $g^{R_2||R_3} \bmod p$ (where $||$ denotes concatenation) and check this value against the Diffie-Hellman public value in the IKE_SA_INIT packet until we find a match.

We measured the time difference between the nonce PRG timestamp and the first key block PRG timestamp and found a mean difference of $154.4\mu s$ with a standard deviation of $32.2\mu s$. We search 3σ out from the average to find the timestamp, requiring a search over 2^8 timestamps.

We also measured the average difference between the first and second calls to the PRG at $18.3\mu s$ with a standard deviation of $4.53\mu s$ for the Fortigate 100D and $1141\mu s$ for the virtual appliance. Measurements were taken using 10 pairs of consecutive calls to the PRG. Since the two key blocks are generated with a single read() system call, we set our search space for each 'second' key PRG block to begin 18 microseconds after the first, searching outwards to a maximum of 32 microseconds after, corresponding to 3σ, or 2^5 timestamps. Combining the simultaneous search for the two timestamps, the key recovery stage requires a search space of $2^8 \cdot 2^5 = 2^{13}$ timestamps.

Since the FortiGate 100D hardware device offloads modular exponentiation to a proprietary Fortigate ASIC (FortiASIC CP8) that uses a transformation we weren't able to reverse-engineer, our brute force code makes a system call to the ASIC to test each candidate pair of PRG outputs. Over 30 trials, the average time to carry out this part of the attack was 3.88s on the hardware.

4.10 Recovering Traffic Keys

Once we have recovered the victim device's public key value, we can make another call to the ASIC with our recovered PRG inputs and the other side's public key exchange value to recover the IKEv2 Phase 1 Diffie-Hellman shared secret. For IKEv2, once the Diffie-Hellman shared secret has been computed, all of the information needed to compute the SKEYSEED value and derive the symmetric encryption keys is present in the clear in the IKE_SA_INIT messages exchanged by both the initiator and responder. We computed the

SKEYSEED as described in Section 2.5.2 and verified full passive decryption against traffic to our FortiGate 100D.

5 MEASUREMENTS

We used ZMap to perform Internet-wide scans on port 443 (HTTPS) and port 500 (IKEv2) to measure the population of vulnerable Fortinet devices. Active scanning is an imperfect measure of the scope of this type of vulnerability. It does not reflect the amount of traffic vulnerable hosts receive. In addition, well-configured hosts would be unlikely to expose either port on a public IP address.

5.1 HTTPS

We used several types of HTTP and HTTPS metadata to identify affected hosts in the wild. Our scans targeted hosts exposing the device's admin panel on a public IPv4 address on port 443.

TLS version and cipher suites. In April 2017 we probed the full public IPv4 address space on port 443 for publicly accessible HTTPS hosts. With each host we performed a TLSv1.0 handshake, the version supported by the vulnerable devices, and offered the cipher suites listed in Table 5 in Appendix B. Our scan completed a full handshake with 29,709,242 hosts. [7]

Server certificate common name. In its default configuration, FortiOS v4 serves a self-signed certificate with the model and serial numbers for the common name and 'Fortinet' for the organization. This does not identify the firmware or build number. We found 114,172 hosts with a matching certificate organization field; their common names indentified 3,379 unique model numbers.

State recovery. Our state recovery attack was successful against 23,517 hosts, or 20.6% of hosts with default Fortinet certificates. We attempted state recovery using a 1s window around the time encoded in the server random. Figure 5 shows the distribution of the number of timestamps guessed for successful state recoveries. In Figure 6, we plot the distribution of the timestamp for the first block of PRG output relative to the timestamp encoded in the TLS server random. The near-uniform distribution may be due to the fact that the server random has second granularity and the PRG uses µsecond granularity.

Figure 4 shows the distribution of the gap between the timestamps for the first and second PRG blocks in the TLS server random. We brute forced up to an offset of 100 μs after the first timestamp, but all our observed state recoveries had a gap of no more than 40 μs between the first and second timestamps.

Specific HTTP files. Our hardware device's administration panel contained an image file located at `/images/logon.gif` that we used as a rudimentary fingerprint for FortiOS. In our HTTPS scan, we sent a GET request for this file. 605,950 hosts responded with HTTP OK, and a corresponding image. The others returned a 404 error. We were unable to automatically validate these images, so we used the techniques below to further narrow candidates.

[7]This is lower than the $\tilde{4}0$ million HTTPS hosts seen in scans offering a wider variety of SSL/TLS versions and cipher suites.

Table 4: X9.31 state and key recovery in the wild

HTTPS hosts (TLS 1.0/port 443)	29,709,242
...with default Fortinet certificate	114,172
...and successful state recovery	23,517
...with known FortiOSv4 ETag	2,336
...and successful state recovery	2,265
IKEv2 hosts (port 500)	7,743,876
...with 128-bit nonces	50,285
...and private key recovery	7
...with TLS nonce state recovery	152
...and non-static IKE parameters	17
...and private key recovery	7

ETag headers. The HTTP ETag header uniquely identifies HTTP server resources, and is used for web cache validation along with conditional requests [27]. The RFC specifies that the value of the header "is data known only to the server". In order to fingerprint devices running vulnerable firmware versions, we matched headers from our scan against known ETags for FortiOS v4.

The Equation Group leak [28] contained a list of 440 ETag suffixes for some FortiOS device and firmware-build pairs, including 168 entries corresponding to 9 models and 26 builds of FortiOS v4. The leak also contained a memory address for each entry, used for the Egregious Blunder exploit with which it was packaged. The ETag for our FortiGate 100D (`5192dbfd`) was not included in the database, so we added it to our search.

Of 655,878 HTTP hosts responding with an ETag, 2,336 gave a known FortiOS v4 ETag. The state recovery attack was successful

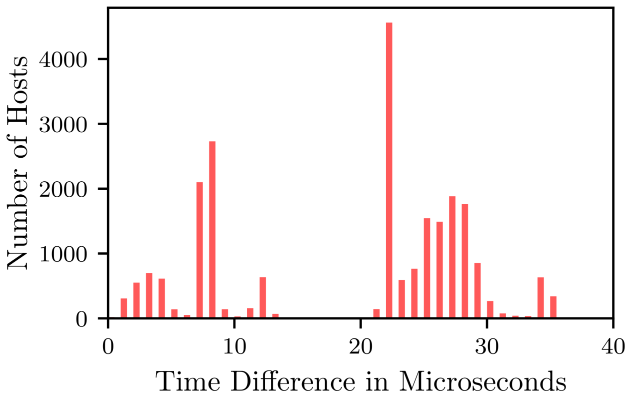

Figure 4: Subsesquent timestamp offset. We calculated the difference between the first and second timestamps used to generate the PRG blocks for the TLS nonce. This value was brute forced from within a range of between zero and one hundred microseconds. The average difference is 19.2 microseconds with a standard deviation of 10.1 microseconds.

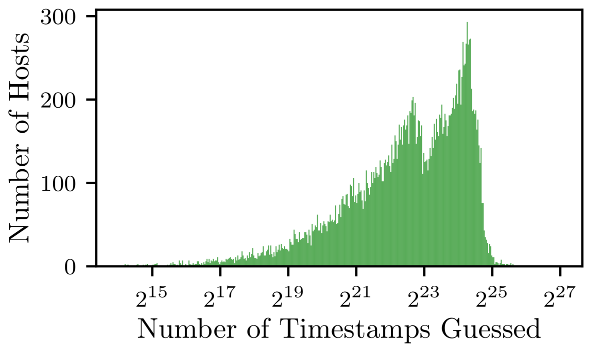

Figure 5: Brute force work for TLS state recovery. The average number of timestamp guesses required for each host was $2^{22.9}$ with a standard deviation of $2^{23.1}$.

for 2,265 (97%) of these. 1,535 of these hosts presented non-default HTTPS certificates. State recovery was successful for every device matching the ETag for our hardware device.

RSA public keys. Fortigate devices were already known to generate RSA moduli that share common factors [32, 34]. We ran a batch GCD computation against the HTTPS certificate RSA public keys from fingerprinted Fortigate devices together with identified Fortigate public keys from historical scans obtained from the authors of [32]. This gave us prime factors for 3,163 keys from our scan. However, the X9.31 state recovery attack described in Section 4.4.1, conducted with the certificate timestamps, was not successful against the most significant bits of the prime factors. None of the hosts with factored keys matched the HTTP or IKE fingerprints, suggesting that certificates were generated by a software version other than FortiOS v4 with a different underlying random number generation vulnerability.

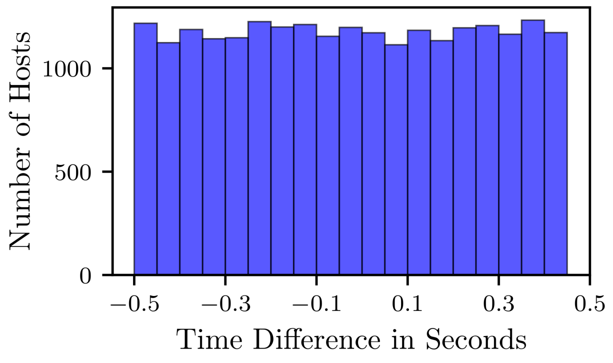

Figure 6: Initial timestamp offset. The average offset between the timestamp encoded in the TLS server random (at 1s resolution) and the timestamp used to seed the first PRG block for our successful state recovery trials (at µs resolution) was 510µs.

Limitations. The devices that responded to our HTTPS scans exposed the administration panel on a public IPv4 address, which is not the default configuration for FortiOS. Our scans therefore identified only those devices that were *misconfigured*. A correct (and default) configuration prevents our scanning but does not mitigate the vulnerability. The total population of vulnerable hosts is therefore likely *significantly* higher than the population visible to our scan. A well-equipped adversary could have constructed a larger database of ETags. This technique can also be used for other manufacturers, a number of whom also insert a model-firmware identifier in header.

5.2 IKEv2

We used UDP scans on port 500 to initiate IKEv2 handshakes for the full IPv4 space. However, the metadata available in IKEv2 connections is more limited than for HTTPS.

HTTPS Admin Panel. Of the 23,554 HTTPS hosts in the previous section against which state recovery from the TLS nonce on port 443 was successful, 152 responded to IKEv2 handshake requests on port 500. Of these hosts, 135 always returned a single, identical, static common nonce and key exchange for every connection. These devices were located within the Chinanet AS and their SSL/TLS certificates suggested a variety of Fortinet model numbers. From the remaining 17 hosts whose nonces and key exchanges were generated on new connections, our key recovery attack succeeded against 7.

Cipher support. We sent handshake requests with cipher proposals that were supported in FortiOS v4, listed in Table 7 in Appendix B. We received 7,743,876 responses.

Nonce size. FortiOS v4's IKEv2 implementation uses 128-bit nonces. From our successful IKEv2 handshakes above, 50,285 had 128-bit nonces. We attempted state and key recovery from our handshakes with all of these hosts, and were able to successfully recover the Diffie-Hellman shared secret in the handshake we negotiated for 7 hosts using the key recovery attack we describe above. This included 4 hosts that weren't seen in the population of vulnerable HTTPS hosts. We hypothesize that most of the publicly visible IKEv2 responders with 128-bit nonces are not vulnerable Fortigate products, and that most VPNs are configured as site-to-site tunnels that would not be visible in our scans.

Limitations. The number of IKE responses we receive should be treated as a lower bound, since many VPNs are configured as site-to-site tunnels, or filter based on source IP and are invisible to scans from unknown hosts.

6 RELATED WORK

Cryptanalysis of RNG designs. There is a long history of cryptanalysis of practical pseudorandom number generator designs in the literature. Kelsey, Schneier, Wagner, and Hall [41] enumerate classes of attacks on PRNGs, and note several design flaws and vulnerabilities against PRNG designs, including the key compromise vulnerability in X9.17/X9.31 RNG that we consider in this paper. Gutterman, Pinkas, and Reinman [30] analyzed the Linux random number generator in 2006, and Dorrendorf, Gutterman, and Pinkas [22] analyzed the Windows random number generator in 2009. Dodis et al. [20] defined a notion of recovery from state compromise for a PRNG, showed that the Linux random number generator did not satisfy this definition, and showed that there were inputs that would cause it to fail to recover from state compromise and would mislead the entropy estimation function. Michaelis, Meyer, and Schwenk [46] analyzed Java random number generation implementations and noted several vulnerabilities, including a vulnerability in Android.

Green [29] notes the dangers of using X9.31. He additionally highlights the danger and usage of a global X9.31 key for the RNG in an early draft of AACS, the digital rights management specification for HD-DVD and Bluray distributions.

Random number generation failures. Multiple types of random number generation failures have been observed in the wild.

One category of RNG failures appears to be due to failure to properly seed a random number generator before use, or seeding with poor-quality inputs. Famously, between 2006 and 2008, the Debian OpenSSL random number generator incorporated almost no entropy into its state. [58] In 2012, Heninger *et al.* discovered a boot-time failure of the Linux random number generator to properly incorporate entropy sources on embedded and headless systems; this flaw resulted in them being able to compute RSA private keys for 0.5% of TLS hosts and DSA private keys for 1.06% of SSH hosts in 2012 [34]. Lenstra et al. [44] performed a similar study of public keys collected from the internet in 2012, and were able to compute RSA private keys for 0.3% of HTTPS hosts and a pair of PGP users. In 2016, Hastings, Fried, and Heninger [33] performed a follow-up study that found low to nonexistent software patching rates for systems affected by the 2012 RNG flaws. Bernstein et al. [10] were able to factor 184 keys from a sample of approximately 2 million smartcard-generated RSA keys from the Taiwanese “Citizen Digital Certificate" smartcard ID system. They hypothesized that the failures were due to a flawed hardware random number generator on some smartcards combined with a failure to whiten raw hardware RNG outputs. Kadianakis et al. [38] performed a similar analysis on the 3.7 million RSA public keys of Tor relays, finding 10 relays with shared RSA moduli and 3,557 relays with shared prime factors.

Other types of system failures can result in repeated states or outputs in RNG implementations. Ristenpart and Yilek [53] show that virtual machine snapshots can result in cryptographic failure due to implementation flaws in random number generators. A 2013 vulnerability in the Android SecureRandom resulted in a number of Bitcoins stolen from Android-based wallets due to repeated DSA signature nonces [42].

Intentional RNG backdoors. A further category of failures are due to intentionally weakened designs. Young and Yung [59] introduced the concept of kleptography, the design of cryptographic schemes with hidden backdoors. They later described a scheme for introducing such a backdoor into discrete log-based cryptosystems [60].

In a 2013 article published on the Snowden leaks, the NY Times and Pro Publica pointed to the NIST-standardized Dual EC DRBG as a cryptographic standard that had been subverted by the NSA as part of a general program to influence standardization processes, although the original source document naming Dual EC has not been published. In the wake of these accusations, NIST removed support for the Dual EC DRBG algorithm from its standards. However, this was not the first time that the possibility of a backdoor in the Dual EC DRBG had been raised. In 2006, Brown [13] noted that the indistinguishability proof for the NIST-standardized Dual EC DRBG relies on a random Q parameter. Shumow and Ferguson [55] noted that the design of the Dual_EC DRBG admits a kleptographic backdoor. By generating parameters such that there exists an integer d where $dQ = P$, the kleptographer can recover the state of the DRBG by observing 32 consecutive bytes of output. Checkoway et al. [16] analyze how an unknown attacker inserted code into Juniper ScreenOS to exploit the presence of the backdoor in the Dual EC DRBG that would allow passive decryption of IPsec connections. Dodis et al. [19] formally model backdoored random number generators, design backdoored PRNGs with strong indistinguishability properties, and evaluate countermeasures against backdoors. Degabriele et al. [18] build on this by giving efficient constructions such PRNGs and bounding the duration of the compromise in terms of the state-size of the PRNG.

7 DISCUSSION

NSA decryption capabilities. Classified NSA documents leaked by Edward Snowden and published by Der Spiegel [61] suggest that the NSA has passive decryption capabilities against some fraction of IPsec, TLS, and SSH traffic. Proposed explanations for these capabilities include the NSA performing 768-bit and 1024-bit discrete log precomputations for widely used Diffie-Hellman primes [6] (Boudot [12] points out that a 768-bit discrete log precomputation may have been feasible for the NSA as early as the year 2000), backdoored random number generation standards such as the Dual EC DRBG [15, 16], and software exploits and malware (“implants”).

We suspect the reality is a combination of these techniques customized to vendors' vulnerabilities. Our paper explores another feasibly exploitable cryptographic vulnerability that may explain some decryption capabilities.

While a compromised random number generator design would seem like an appealing avenue to inject or discover vulnerabilities in cryptographic implementations, the Dual EC DRBG just does not seem to have been implemented widely enough to explain decryption capabilities in more than a small handful of products. (The exceptions we are aware of are the RSA BSAFE library, and Juniper ScreenOS.) By contrast, the X9.17/X9.31 PRG has been ubiquitous for decades.

Ease of exploitation. We note that our attacks in this paper against the X9.31 PRG were significantly less computationally expensive to carry out than many of the attacks against Dual EC in TLS measured by Checkoway et al. [16]. This is because the most efficient attacks against Dual EC require 32 bytes of raw PRG output, and the effort required to exploit the backdoor grows exponentially as the amount of raw PRG output available to the attacker decreases. In contrast, because successive timestamps do not have very much entropy, an efficient attack against the X9.31 PRG with AES for the block cipher that uniquely recovers the state would be possible with 20 bytes or even fewer of raw output. Checkoway et al. [16] note that when Juniper replaced the X9.31 PRG with Dual EC in their ScreenOS implementation, they increased the length of the nonces used in the IKE handshake from 20 bytes to 32 bytes, thus permitting efficient passive exploitation of the Dual EC backdoor. Efficient Dual EC exploitation would not have been possible without this increase.

NOBUS and symmetric backdoors. As we note in the introduction, the vulnerability we exploit in the X9.17/X9.31 PRG is by definition not a "NOBUS" backdoor because it is symmetric, and is thus both detectable and exploitable by any party who can gain access to a static key used by some device for the PRG through reverse-engineering or physical access. This is in contrast to the case of the Dual EC PRG, where only the party who generated the elliptic curve points used as parameters for the PRG knows whether they contain a backdoor. However, an implementation of the X9.17/X9.31 PRG that uses a vulnerable static key could still increase the cost of exploitation to a chosen level of difficulty by increasing the granularity of the timestamps. The Fortinet systems we analyzed used `gettimeofday` which typically has at most μs resolution. An implementation using `RDTSC` to obtain nanosecond granularity instead, would likely have put the attack outside easy reach of modest attackers.

Failure of the standardization process. The failure of the NIST and FIPS standardization process to protect against a long-known vulnerability in an approved random number generator is surprising. The observation that the seed key must remain secret in the X9.17/X9.31 design was first noted almost two decades ago, and yet none of the descriptions of the algorithm we could find mentioned the importance of generating an unpredictable key. The security policies documenting a known vulnerability should have been detected by the testing labs; the fact that they were not illustrates systemic issues with lab-based validation. NIST mentioned conflict of interest issues (the testing labs are paid by the vendors), lab personnel skill, and workload in personal communication to us. To address these issues, NIST is transitioning to an automated validation program [49].

Eliminating obsolete cryptography. John Kelsey, one of the authors of [41], told us in personal communication that removing X9.17 key generation and FIPS 186's RNGs from the standards that ultimately became NIST SP 800-90A was one of the first things he did when joining NIST, and that he was surprised to learn in 2016 that implementations using both remained in the field.

Removing obsolete cryptographic algorithms from standards and implementations is difficult in practice. The MD5 and SHA1 hash functions, RC4 stream cipher, and RSA PKCS#1v1.5 encryption padding remained in use for decades after they were known to be cryptographically flawed. For vendors, removing algorithms breaks backwards compatibility and many devices have long lifespans. Kelsey pointed to the difficulty of eliminating obsolete cryptography as a contributing factor to the vulnerability; we hypothesize that once the X9.31 PRG was en route to deprecation, there was little incentive for NIST to update the standard, but vendors continued to implement the algorithm as standardized for many years because of the long, slow deprecation process. Standards such as FIPS may also increase the cost of updating cryptography by necessitating expensive new product certification.

Concerns about other PRG designs. In positive news, the remaining approved PRG designs in NIST SP 800-90A appear to be based on sounder footing, both in practice and in theory. However, this analysis assumes that implementations are sound. Cipher-based PRGs appear specifically vulnerable to state recovery attacks when the cipher key is obtained by an attacker. This raises the possibility that a careless or malicious implementation of a modern PRG such as NIST's CTR_DRBG [8] could be implemented in such a way that the key is *not* routinely updated, which might allow state recovery attacks. These attacks are problematic, as an observer without knowledge of the key would see output that is statistically indistinguishable from a correct implementation [36]. Moreover, such vulnerabilities might not be visible in test modes due to implementation differences [45].

ACKNOWLEDGEMENTS

We thank David McGrew and Dario Ciccarone for helpful discussions and research into Cisco's product lines, and Steve Checkoway for reverse-engineering the Juniper ScreenOS implementation of the X9.31 PRG. This work was supported by the National Science Foundation under grants CNS-1651344, CNS-1505799, CNS-1408734, CNS-1010928, CNS-1228443, CNS-1653110 and EFMA-1441209; the Mozilla Foundation; and a gift from Cisco. We are grateful to Cisco for donating the Cisco UCS servers we used for the computational experiments.

REFERENCES

[1] [n. d.]. Cryptographic Algorithm Validation Program - RNG Validation List. https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/validation/validation-list/rng.

[2] [n. d.]. FIPS 140-2 SECURITY POLICY FOR: INZERO GATEWAY. www.kmip.me/www3.cryptsoft.com/fips140/unpdf/140sp1841-1.html.

[3] IDC Corporation. [n. d.]. Worldwide Security Appliance Market Off to a Healthy Start in 2016, Continuing Its Streak of Eleven Consecutive Quarters of Growth, According to IDC. https://www.idc.com/getdoc.jsp?containerId=prUS41490016

[4] MITRE Corporation. [n. d.]. CVE-2016-1909. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1909

[5] MITRE Corporation. [n. d.]. CVE-2016-8492. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8492

[6] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *22nd ACM Conference on Computer and Communications Security.*

[7] Elaine Barker and Allen Roginsky. 2011. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication* 800 (2011), 131A.

[8] Elaine B Barker and John Michael Kelsey. 2007. *Recommendation for random number generation using deterministic random bit generators (revised).* US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory.

[9] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. 2014. *Security of Symmetric Encryption against Mass Surveillance.* Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-44371-2_1

[10] Daniel J Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. 2013. Factoring RSA keys from certified smart cards: Coppersmith in the wild. In *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 341–360.

[11] Manuel Blum and Silvio Micali. 1984. How to Generate Cryptographically Strong Sequences of Pseudo-random Bits. *SIAM J. Comput.* 13, 4 (Nov. 1984), 850–864. https://doi.org/10.1137/0213053

[12] Fabrice Boudot. 2017. On Improving Integer Factorization and Discrete Logarithm Computation using Partial Triangulation. Cryptology ePrint Archive, Report 2017/758. http://eprint.iacr.org/2017/758.

[13] Daniel RL Brown. 2006. Conjectured Security of the ANSI-NIST Elliptic Curve RNG. *IACR Cryptology ePrint Archive* 2006 (2006), 117.

[14] Ed. C. Kaufman. 2005. Internet Key Exchange (IKEv2) Protocol. IETF RFC RFC4306.

[15] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. 2014. On the Practical Exploitability of Dual EC in TLS Implementations. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14).* USENIX Association, Berkeley, CA, USA, 319–335. http://dl.acm.org/citation.cfm?id=2671225.2671246

[16] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. 2016. A systematic analysis of the Juniper Dual EC incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 468–479.

[17] M. Schneider D. Maughan, M. Schertler and J. Turner. 1998. Internet Security Association and Key Management Protocol. IETF RFC RFC2408.

[18] Jean Paul Degabriele, Kenneth G. Paterson, Jacob C. N. Schuldt, and Joanne Woodage. 2016. Backdoors in Pseudorandom Number Generators: Possibility and Impossibility Results. In *Advances in Cryptology – CRYPTO 2016*, Matthew Robshaw and Jonathan Katz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 403–432.

[19] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. 2015. *A Formal Treatment of Backdoored Pseudorandom Generators.* Springer Berlin Heidelberg, Berlin, Heidelberg, 101–126. https://doi.org/10.1007/978-3-662-46800-5_5

[20] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. 2013. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM, 647–658.

[21] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. 2014. How to Eat Your Entropy and Have It Too – Optimal Recovery Strategies for Compromised RNGs. In *CRYPTO '14.*

[22] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. 2009. Cryptanalysis of the random number generator of the Windows operating system. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 10.

[23] Digital Signature Standard (DSS). 2013. Generation of Probable Primes with Conditions Based on Auxiliary Probable Primes. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[24] Fortinet. [n. d.]. FG-IR-16-067: FortiOS local privilege escalation via malicious use of USB storage devices. http://fortiguard.com/psirt/FG-IR-16-067.

[25] Fortinet. [n. d.]. FG-IR-17-245: DUHK Attack against Fortinet Products. https://fortiguard.com/psirt/FG-IR-17-245.

[26] Fortinet. 2016. Brief Statement Regarding Issues Found with FortiOS. https://web.archive.org/web/20160125202411/http://blog.fortinet.com:80/post/brief-statement-regarding-issues-found-with-fortios.

[27] John Franks, Phillip M. Hallam-Baker, Jeffery L. Hostetler, Scott D. Lawrence, Paul J. Leach, Ari Luotonen, and Lawrence C. Stewart. 1999. *HTTP Authentication: Basic and Digest Access Authentication.* RFC 2617. RFC Editor. http://www.rfc-editor.org/rfc/rfc2617.txt http://www.rfc-editor.org/rfc/rfc2617.txt.

[28] Dan Goodin. 2016. Group claims to hack NSA-tied hackers, posts exploits as proof. https://arstechnica.com/information-technology/2016/08/group-claims-to-hack-nsa-tied-hackers-posts-exploits-as-proof/

[29] Matthew Green. 2016. Random number generation: An illustrated primer. https://blog.cryptographyengineering.com/2012/02/21/random-number-generation-illustrated/

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

[31] D. Harkins and D. Carrel. 1998. The Internet Key Exchange (IKE). IETF RFC RFC2409.

[32] Marcella Hastings, Joshua Fried, and Nadia Heninger. 2016. Weak keys remain widespread in network devices. In *Proceedings of the 2016 Internet Measurement Conference.* ACM.

[33] Marcella Hastings, Joshua Fried, and Nadia Heninger. 2016. Weak keys remain widespread in network devices. In *Proceedings of the 2016 ACM on Internet Measurement Conference.* ACM, 49–63.

[34] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium.*

[35] William Herlands, Thomas Hobson, and Paula Donovan. 2014. Effective Entropy for Memory Randomization Defenses. In *USENIX 7th Workshop on Cyber Security Experimentation and Test.* Lincoln Labratory.

[36] Dan Shumow Joanne Woodage. 2018. An Analysis of the NIST SP 800-90A Standard.

[37] Juniper Networks, Inc. [n. d.]. FIPS 140-2 SECURITY POLICY - SSG 140. https://www.juniper.net/documentation/hardware/netscreen-certifications/Security_Policy_SSG-140_ScreenOS_6_2.pdf.

[38] George Kadianakis, Claudia V Roberts, Laura M Roberts, and Philipp Winter. [n. d.]. "Major key alert!" Anomalous keys in Tor relays. ([n. d.]).

[39] Sharon S. Keller. 2005. NIST-Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms. National Institute of Standards and Technology.

[40] John Kelsey, Bruce Schneier, and Neils Ferguson. 1999. Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In *SAC '99.*

[41] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption.* Springer, 168–188.

[42] Alex Klyubin. 2013. Some SecureRandom Thoughts. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html.

[43] Sharon S. Keller Lawrence E. Bassham III. 2005. The Random Number Generator Validation System (RNGVS). National Institute of Standards and Technology.

[44] Arjen Lenstra, James P Hughes, Maxime Augier, Joppe Willem Bos, Thorsten Kleinjung, and Christophe Wachter. 2012. Public Keys. In *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417.* 626–642.

[45] Steve Marquess. 2013. Flaw in Dual EC DRBG (no, not that one). http://marc.info/?l=openssl-announce&m=138747119822324&w=2.

[46] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. 2013. Randomly Failed! The State of Randomness in Current Java Implementations.. In *CT-RSA*, Vol. 7779. Springer, 129–144.

[47] Stephan Müller. [n. d.]. Linux Random Number Generator — A New Approach. Available at http://www.chronox.de/lrng/doc/lrng.html.

[48] Phong Q Nguyen and Igor E Shparlinski. 2003. The insecurity of the Elliptic curve Digital Signature Algorithm with partially known nonces. *Designs, codes and cryptography* 30, 2 (2003), 201–217.

[49] NIST. [n. d.]. Automated Cryptographic Validation Testing. https://csrc.nist.gov/Projects/Automated-Cryptographic-Validation-Testing.

[50] NIST. 2001. SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf.

[51] NIST. 2017. CMVP Historical Validation List. http://web.archive.org/web/20170120035228/http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-historical.htm.

[52] Nicole Perlroth. 2013. Government Announces Steps to Restore Confidence on Encryption Standards. Available at https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/?_r=1. *The New York Times* (2013).

[53] Thomas Ristenpart and Scott Yilek. 2010. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *NDSS '10.*

[54] Sylvain Ruhault. 2017. SoK: Security Models for Pseudo-Random Number Generators. In *IACR Transactions on Symmetric Cryptography (TOSC)*, Vol. 1.

[55] Dan Shumow and Niels Ferguson. [n. d.]. On the possibility of a Back Door in the NIST SP800-90 Dual EC PRNG. http://rump2007.cr.yp.to/15-shumow.pdf

[56] Falko Strenzke. 2016. An Analysis of OpenSSL's Random Number Generator. In *EUROCRYPT '16.* Springer-Verlag New York, Inc., New York, NY, USA, 644–669. https://doi.org/10.1007/978-3-662-49890-3_25

[57] Wladimir J van der Laan. 2016. TADAQUEOUS moments. http://laanwj.github.io/2016/09/01/tadaqueos.html

[58] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. 2009. When Private Keys are Public: Results From the 2008 Debian OpenSSL Vulnerability. In *Proceedings of IMC 2009*, Anja Feldmann and Laurent Mathy (Eds.). ACM Press, 15–27.

[59] Adam Young and Moti Yung. 1997. Kleptography: Using cryptography against cryptography. In *Eurocrypt*, Vol. 97. Springer, 62–74.

[60] Adam Young and Moti Yung. 1997. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In *Annual International Cryptology Conference.* Springer, 264–276.

[61] zzz intro to the vpn 2010. Intro to the VPN Exploitation Process. Media leak. http://www.spiegel.de/media/media-35515.pdf.

A FORTIOS V4 X9.31 INITIALIZATION ROUTINE

Listing 1: The X9.31 Initialization Routine.

```c
int initialize_X931()
{
  char rng_state[16];
  char timestamp_buffer[16];
  int aes_key[4];
  int result = key_set;
  aes_key[0] = 0x6D66B1F3;
  aes_key[1] = 0x42726013;
  aes_key[2] = 0xAB1C06ED;
  aes_key[3] = 0x0262D4B8;
  if ( !key_set )
    result = set_aeskey(aes_key);
  if ( !state_set )
  {
     /* initial state setting removed for
        clarity */
    save_state(rng_state);
    fill_timestamp(timestamp_buffer);
    result =
       x931(&timestamp_buffer, output_buffer,
                   rng_state, 16);
  }
  return result;
}
```

B SUPPORTED CIPHER SUITES IN FORTIOSV4

Our hardware device supported the following cipher suites. Our scanning client used in Section 5 offered all of these cipher suites.

Table 5: Supported TLS Cipher Suites in FortiOS v4

TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_RSA_WITH_RC4_128_SHA
TLS_RSA_WITH_RC4_128_MD5

Table 6: Supported IKEv1 Parameters in FortiOS v4

Cipher	PRF	Group	Authentication
DES	MD5	DH_768	PSK
3DES	SHA1	DH_1024	RSA
AES-128	SHA256	DH_1536	
AES-192			
AES-256			

Table 7: Supported IKEv2 Parameters in FortiOS v4

Cipher	PRF	MAC	Group
DES	SHA256	SHA256	DH_768
3DES	SHA1	SHA1	DH_1024
AES-128	MD5	MD5	DH_1536
AES-192			DH_2048
AES-256			