Practical state recovery attacks against legacy RNG implementations

Shaanan Cohney*, Matthew D. Green†, Nadia Heninger*
*University of Pennsylvania shaanan@cohney.info,nadiah@cis.upenn.edu
†Johns Hopkins University mgreen@cs.jhu.edu

***Abstract*—The ANSI X9.17/X9.31 random number generator is a pseudorandom number generator design based on a block cipher and updated using the current time. First standardized in 1985, variants of this PRNG design were incorporated into numerous cryptographic standards over the next three decades. It remained on the list of FIPS 140-1 and 140-2 approved random number generation algorithms until January 2016. The design uses a static key with the specified block cipher to produce pseudo-random output. It has been known since at least 1998 that the key must remain secret in order for the random number generator to be secure. However, neither the FIPS 140-2 standardization process in 2001 or NIST's update of the algorithm in 2005 appear to have 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 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 develop a full passive decryption attack against FortiGate VPN gateway products using FortiOS version 4. Private key recovery requires a few seconds of computation. We measured the prevalence of this vulnerability on the visible Internet using active scans and find that we are able to recover the random number generator state for 21% of HTTPS hosts serving a default Fortinet product certificate, and 97% of hosts with metadata identifying FortiOSv4. We successfully demonstrate full private key recovery in the wild against a subset of these hosts that accept IPsec connections.

I. 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 [1]; to catastrophic vulnerabilities that allow for adversarial recovery of all of random coins used in a protocol execution [2], [3]. 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 malicious backdoor [4]. 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 [3].

The problem of constructing random and pseudorandom number generators has been extensively explored by industry [5], [6], [7] and in the academic literature [8], [9], [10], [11], [12]. 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,[1] and prior to 1998 only one such algorithm was specified. Recent discoveries surrounding the inclusion of flawed generators motivate a more thorough examination of these generators — and particularly, their use in approved cryptographic products.

The ANSI X9.17/31 standards. The ANSI X9.17 "Financial Institution Key Management (Wholesale)" standard, first published in 1985, defined a voluntary interoprability standard for cryptographic key generation and distribution for the financial industry. This standard included a pseudorandom number generator (PRG) in 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.

This random number generator design appeared in government cryptographic standards for the next two decades, occasionally updated with new block cipher designs. 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

[1]Maintained by the U.S. National Institute of Standards and Technology (NIST), the FIPS 140 series of documents outline standards for validating cryptographic modules. This standard is used by the U.S. Cryptographic Module Validation Program to perform official certification of products used in U.S. government applications, including banking.

published extensions of this design using three-key 3DES and AES as the block cipher [13] 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 the key should become known, an attacker can use the key to decrypt the output and recover *all future and past states* of the random number generator by brute forcing the timestamp [8]. 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 [8], the NIST documents specifying the ANSI X9.31 PRG design fail to specify any requirements for how the cipher key should be generated [13]. 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 non-trivial subset 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 to generate encryption keys for IPsec uses a hard-coded key, which is a test value given in the NIST RNGVS specification [14], published as a validation suite alongside their standardization of the generator. We are able to 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 random number generator. However, the failure mode of static, discoverable keys we exploit was not ruled out by standards, and appears to have been independently implemented by a variety of vendors attempting to deploy this random number generator. This is a failure of the standardization process that has led to real and ongoing vulnerabilities.

A. Our Contributions

- To our knowledge, we are the first to note that the official standardized descriptions of the X9.17/X9.31 fail to protect implementers against state compromise attacks.
- We perform a systematic study of FIPS 140 security policy documentation and discover several independently vulnerable RNG implementations from different vendors.
- We develop an efficient passive X9.31 state recovery attack for the FortiOS v4 IPsec implementation and demonstrate full IPsec VPN decryption.
- We use Internet-wide measurements to measure the scope of this vulnerability among publicly visible hosts, and demonstrate exploitation on real hosts in the wild.

B. Disclosure

We disclosed the X9.31 vulnerability to Fortinet in October 2016. Fortinet responded by releasing a patch for affected versions of FortiOS. FortiOS version 5 did not implement the X9.31 PRG and is not vulnerable.

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

We notified the remaining vendors listed in Table I 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 already disallowed use of the ANSI X9.31 RNG independently of our work. Despite these disclosures, we detected many vulnerable devices still active on the open Internet, and additional devices may reside within enterprise networks.

C. 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 attack 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 also 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 to their networks.

II. Background

A. Pseudorandom generators

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 $\mathsf{out}^{\mathsf{q}}(\mathsf{G}, s_0)$ 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 $\mathsf{out}^{\mathsf{q}}$ and a set of random bits.[2]

The PRG discussed in this work extends this basic definition slightly, as the generate function G also takes (and may return) some *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.

B. 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 giving 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 updated versions of the design using three-key 3DES and AES in 2005. [13] While this design has appeared under various names for the past three decades, 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. In our case of interest, we will specialize to AES, and define $\ell = 128$. $E_K(Y)$ represents the encipherment of Y under the 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 [13] 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

$$V_i = E_K(R_i \oplus I_i)$$

[2]We draw our notation from the definition of Dodis et al. [15].

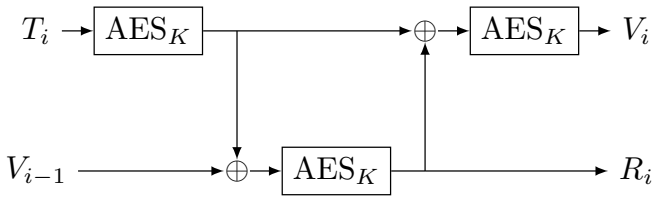

Fig. 1: A single iteration of the ANSI X9.31 PRG generation function (G) using AES as the symmetric cipher. The inputs are a timestamp T_i and a seed V_{i-1}. The iteration produces an output block R_i and a new seed V_i on each iteration.

The output of G is the string $\mathsf{truncate}_n(R_1 \| R_2 \| \ldots \| R_b)$ 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.

C. 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 (PRP). Kelsey et al. [8] observed that the generator is clearly vulnerable when K is not kept secret. An attacker who is able to learn K can recover the current state using two consecutive blocks of output together with guesses for the timestamps used to generate them. (A single block of output is not enough to uniquely identify the state, but two blocks will almost surely identify it uniquely.) Let R_0 be a block of output generated at T_0, and R_1 be a block of output generated at T_1, and let $D(Y)$ be the decryption of Y using key K. We can rearrange the above equations to write the first block in terms of the second block and the timestamps:

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

If the timestamps are only known approximately, we can brute force the timestamps within some range until we find a pair that yields equality, or apply a meet-in-the-middle attack [8]. 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 two timestamps T_1 and T_2 have been recovered, the seed for the next round is

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

A guess for the output from the next iteration of the random number generator is then uniquely defined by a guess for the following 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 to recover the state. With this state in hand, the attacker can now predict any subsequent output by running the normal generation algorithm using a guess for each subsequent timestamp. Alternatively, she can recover *previous* output blocks by "winding the generator backwards" using a

guess for earlier timestamps. Both attacks require the same effort.

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

D. 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: (1) where the Dual EC backdoor was 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, the X9.31 vulnerability is symmetric, and any implementation that stores a fixed secret key in code or hardware is vulnerable to passive exploitation by an attacker who can recover the key through reverse engineering. Separately (2) the Dual EC attack requires a minimum of approximately 28 bytes of contiguous PRG output to perform a successful attack, while the ANSI attack can be conducted with fewer bytes.[3] Surprisingly, this second restriction can play a major role in determining in the cost of an attack on a protocol such as TLS or IPsec.

1) TLS Background: A TLS 1.0, 1.1, or 1.2 handshake begins with a client hello message, which contains a 32-byte random nonce and a list of supported cipher suites. The server responds with a server hello message, which contains a 32-byte random nonce, the server's choice of cipher suite, and the server's certificate which contains its long-term public key. The server and client then negotiate shared secret keying material using the asymmetric cipher chosen by the server. In the case of RSA, the client encrypts a secret to the server's public key; in the case of Diffie-Hellman or elliptic curve Diffie-Hellman, the server and client exchange Diffie-Hellman key exchange messages. At this point the client and server authenticate the handshake using symmetric keys derived from their negotiated shared secret and the nonces, and swich to sending symmetrically encrypted data.

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 since the two timestamps are likely generated in quick succession, the attacker will likely recover a unique possible state. For a Diffie-Hellman based key exchange, this attack could work if either the client or server uses the vulnerable PRG; for RSA key exchange, the client generates the RSA-encrypted premaster secret and the key exchange would only be compromised if the client uses the vulnerable PRG.

[3]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 (approximately $M * 2^{-(128-n)}$ when brute forcing over a timestamp space of size M) even when n is large.

E. Attacking X9.31 in IPsec

Checkoway *et al.* [3] 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 decryption.

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 the implementation of Virtual Private Networks (VPNs). The IKE (Internet Key Exchange) family of protocols allows 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 versions use the Diffie-Hellman key exchange protocol to establish a shared secret.

IKEv1. The original IKE specification [17] 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. In Main Mode, the initiator sends a Security Association (SA) payload, with a series of proposals for combinations of cipher suites and parameters. The responder responds with an SA payload indicating its chosen cipher proposal. Each party then sends a Key Exchange (KE) message, which contains a Diffie-Hellman key exchange payload, together with information needed to authenticate the key exchange. The format and the information differs depending on the chosen authentication method. IKE supports four authentication methods: two public key encryption methods, a digital signature method, and a pre-shared key authentication mode. When using digital signatures or a pre-shared key to authenticate, both the initiator and responder send their Diffie-Hellman key exchange message together with a cleartext nonce of length between "8 and 256 bytes inclusive" [17]. Each packet in the exchange includes an 8-byte (non-random) cookie[4] from the initiator and responder, which is used to uniquely identify the connection to each participant.

Both parties then compute a series of symmetric encryption and authentication keys from the Diffie-Hellman shared secret, the nonces, and the cookies. If PSK authentication is used, the PSK is also incorporated into this key derivation process. All messages following this point are now encrypted using these newly generated symmetric keys. Each side then exchanges 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

[4]The ISAKMP specification (RFC 2408) [18] suggests that the cookie be generated by applying the MD5 hash function to the participant IPs, ports, and a local random secret. It does not consist of raw RNG output.

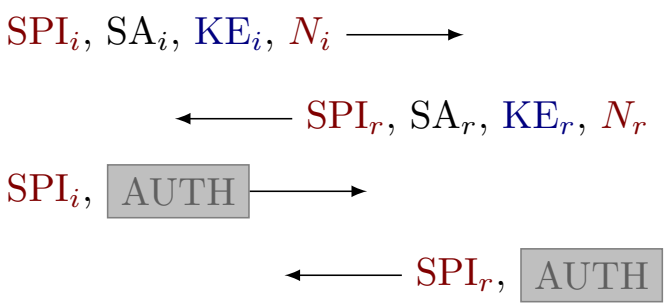

Fig. 2: **Randomness and the IKEv2 Handshake.** The IKEv2 handshake establishes an authenticated, encrypted connection using a Diffie-Hellman key exchange. In our target implementation, both the SPI and nonce N are raw, unencrypted outputs from the PRG. We will use them to recover the secret used to generate the key exchange message KE, which is generated from the PRG immediately afterward. The encrypted portions of the handshake are inside of a gray box.

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 [19]. IKEv2 eliminates the two-phase structure from the IKE specifications and a handshake can be completed in as few as four messages. 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 header containing a cookie called the SPI. [5]

The responder replies with its own IKE_SA_INIT messages either containing its public key if the initiator guessed correctly and otherwise containing an INVALID_KE_PAYLOAD and causing the initiator to retry with "the corrected Diffie-Hellman Group" [19].

After a completed Diffie-Hellman key exchange, the two parties 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 optionally can perform a second Diffie-Hellman key exchange.

[5]In IKEv2's initial key exchange phase the cookie field from IKEv1 is renamed to the 'Security Parameter Index' (SPI) not to be confused with the IPSec SPI that identifies a particular SA. Furthering confusion, the IKEv2 specification also defines a COOKIE SA payload, designed to help thwart resource exhaustion attacks, this was known as the IPSec SPI in IKEv1.

2) State recovery in IPsec: An attack on the IKE handshake exploiting a vulnerable X9.31 implementation would proceed much as described in [3]. The attacker would need raw X9.31 output to be used to generate both the random nonce and the secret key used for the Diffie-Hellman key exchange for either initiator or responder, the nonce should be close to two block lengths long, and ideally the Diffie-Hellman secret key would be generated shortly after the nonce. The attacker would then guess the timestamps used to generate the nonce in order to recover the state of the random number generator at each step. The attacker can confirm a correct guess by checking equality for Equation 2. The attacker then guesses the two timestamps used for the next two blocks of output using Equation 3 and confirms the guesses by calculating the Diffie-Hellman public value (using the two output blocks as the exponent) and comparing against the intercepted value.

Full symmetric key recovery for IKEv1 depends on the details of the authentication method. The attacker can validate state recovery and Diffie-Hellman secret compromise against a single key exchange packet from one side of the connection, but depending on the authentication type they may need further information to generate the session keys. For pre-shared key authentication, the attacker would need to learn the pre-shared key in addition to the nonces and cookies that appear in the clear in the handshake. 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 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 in the clear: timestamps, nonces, the SPI nonce, and both key exchange values. We note that IKEv2 with a PSK uses the PSK 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.

III. Hardcoded X9.31 Keys in FIPS Certification

As discussed in Section II-B, the NIST design description for the X9.31 random number generator [13] 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 certfiying usage the X9.31 random number generator made available on the NIST web site [20] to understand how many vendors publicly documented static or hard-coded keys that may render them vulnerable to a state recovery attack.

A. Background on FIPS certification

FIPS Publication 140-2 "Security Requirements For Cryptographic Modules" [21] defines requirements for devices and software that implement cryptography. FIPS 140-1 was originally released July 17, 1995, establishing the program, and was superseded on May 25, 2001 by FIPS 140-2. Compliant devices are eligible for certification under the Cryptographic Algorithm Validation Program (CAVP), administered by NIST, and The Cryptographic Module Validation Program (CMVP) for which the CAVP is a prerequisite, administered by NIST and the Communications Security Establishment (CSE) of Canada. Once a device has been certified under the CMVP, it is added to a list of approved devices that federal agencies and other regulated bodies are allowed to use.

The annexes to FIPS 140-2 lists approved algorithms for each of "Security Functions", "Protection Profiles", "Approved Random Number Generators" and "Key Establishment Techniques". 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 published an advisory, NIST SP 800-131A, announcing the transition away from smaller key lengths and weaker cryptographic algorithms, including the impending deprecation of the X9.31 RNG [22]. Currently, the only approved random number generators approved for use are those listed in NIST SP 800-90A, which was updated in June 2015 to remove Dual EC DRBG.

B. 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 provdes a list of implementations certified for historical random number generators.[6] 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 certificates include a list of Critical Security Parameters (CSP), which include access control, key and parameter generation, and zeroization policy. We also looked for language elsewhere in the documentation mentioning the source of the seed key and seed. 127 of the vendors did not mention the AES key in the list of CSPs or elsewhere in the documentation.

[6]https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/validation/validation-list/rng

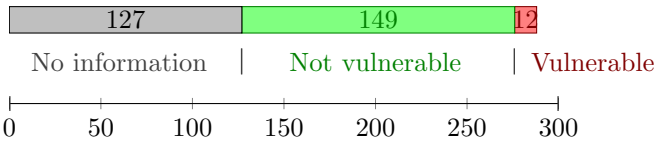

Fig. 3: **Counting vulnerable implementations.** We examined the security policy documents from 288 vendors who had been FIPS 140-2 certified for the X9.31 PRG for information on how the seed key for the random number generator was generated. For 44% of cases, there was no information on key generation; for 52% the documents mentioned generating the keys randomly, and the remaining 12 vendors, 4% of the total, documented vulnerable behaviors such as hard-coded keys.

Since we are unable to determine whether the key was generated securely, we exclude these from further study. This left 161 vendors who did 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 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, instantiated as `/dev/urandom`. As an example of language indicating what we considered to be safe X9.31 key generation, the InZero Gateway XB2CUBSB3.1 security policy states that the "PRNG is seeded from `/dev/urandom`...this provides the PRNG with 256 bits of entropy for the seed key" [23]. 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 [24], 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, 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 vulnerable to a state recovery attack if the documentation stated that a single key was used for the lifetime of a device, particularly if 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 I.

Vendor	Product Line	Version	Language Used	X9.31 Removed
BeCrypt Ltd.	BeCrypt Cryptographic Library	2.0	"Compiled into binary"	3.0
Cisco Systems Inc	Aironet	7.2.115.2	"statically stored in the code"	v8.0
Deltacrypt Technologies Inc	DeltaCrypt FIPS Module		"Hard Coded"	N/A
Fortinet Inc	FortiOS v4	4.3.17	"generated external to the module"	4.3.18
MRV Communications	LX-4000T/LX-8020S	v5.3.8	"Stored in flash"	v5.3.9
Neoscale Systems Inc	CryptoStor	2.6	"Static key, Stored in the firmware"	
Neopost Technologies	Postal Security Devices	v28.0	"Entered in factory (in tamper protected memory)"	v30.0
Renesas Technology America	AE57C1	v2.1012	"With the exception of DHSK and the RNG seed, all CSPs are loaded at factory."	
TechGuard Security	PoliWall-CCF	v2.02.3101	"Generation: NA/Static"	
Tendyron Corporation	OnKey193	v122.102	"Embedded in FLASH"	
ViaSat Inc	FlagStone Core	v2.0.5.5	"Injected During Manufacture"	
Vocera Communications Inc.	Vocera Cryptographic Module	v1.0	"Hard-coded in the module"	v2.0

TABLE I: FIPS 140-2 Security Policies Documenting X9.31 Implementations Potentially Vulnerable to State Recovery. 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.

C. Device-specific analysis

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

We contacted Cisco about X9.31 usage, and they confirmed that X9.31 was used in Aironet 12.4-based branches for access points and WISM modules 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 has also 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 also told us that they are not currently shipping any products that use 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. When we contacted BeCrypt, they told 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, the weak entropy input is actually strong entropy and the key is generated programmatically at startup" and additionally that the only fixed key relating to a RNG 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 tamperproof memory would be quite difficult to attack.

D. 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.

IV. 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. An RNG 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.

A. History of FortiOS 4.x

FortiOS is a network operating system created by Fortinet Inc. for their network security hardware devices and virtual appliances. As of 2016, they were the fourth largest vendor by market share according to the IDC Worldwide Quarterly Security Appliance Tracker [25]. Fortigate primarily specializes in firewalls, intrusion detection systems and VPN gateways. FortiOS is used widely across their product suite except for client-side software packages.

FortiOS 4.0 was first released on February 20, 2009 for a limited set of devices. Fortnet released three major versions, supporting a number of them concurrently while sunsetting older major versions. FortiOS 4.1 was released February 8, 2009, 4.2 in March 2010, and 4.3 on March 18, 2011. The

X9.31 RNG was implemented in FortiOS version 4.3 (2011) but is not present in FortiOS 5, released in November 2012.

Prior to our disclosure of the PRNG 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 [26], Fortigate released version 4.3.19 of FortiOS in November 2016.

1) Vulnerabilities in FortiOS: On January 15, 2016, the MITRE corporation posted CVE-2016-1909 [27] 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 [28], 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 issue 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 APT (advanced persistent threat) actor known as "The Equation Group". Among other things, the leak contained a remote code execution exploit for FortiOS v3 and v4 titled EGREGIOUSBLUNDER. The documentation for the exploit included information on identifying FortiOS versions based on HTTP 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 entrypoint to FortiOS's X9.31 implementation [29]. 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.

B. Static Analysis

We analyzed two implementations of FortiOS v4, the embedded operating system for Fortigate's network devices. The first was firmware obtained from a filesystem and memory dump of 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 V.

FortiOS is a GNU/Linux variant, with a customized shell providing configuration capabilities and kernel modules implementing hardware interfaces and cryptographic functions. The kernel is Linux 2.4.37, the last release of the 2.4.x series kernels released in December 2008, which reached end of life in December 2011. While kernel 2.6.32 was available as of the release of FortiOS v4, embedded device manufacturers commonly chose to use the earlier kernels due to potential performance improvements and incompatibilities in the loadable kernel module (LKM) system. FortiOS v5 continues to use the Linux 2.4.37 kernel.

Each firmware release contains files for symmetric multiprocessor systems, such as the FortiGate 100D we analyzed, and for single processor systems.

C. 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 node corresponding to the X9.31 character device.

We reverse-engineered the kernel module providing the X9.31 implementation in Listing 1 in Appendix A and found the hard-coded AES key used for the RNG. The hard-coded key was `f3b1666d13607242ed061cabb8d46202`. The key used in both the firmware dump and virtual appliance was the same. 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 [14] and appears to have been hard-coded into the source code.

The PRNG implementation generates the timestamp 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 input timestamp to the X9.31 generator.

D. 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.

The TLS server hello random consists of a four-byte timestamp followed by two raw blocks of X9.31 RNG 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 because it uses an OpenSSL-based daemon and uses the OpenSSL RAND_bytes() pseudo-random number generator to generate the secret keys used in the TLS handshake. The system X9.31 generator is used only to generate the initial seed for the OpenSSL PRNG and to generate the server hello random nonce.

E. The IKE Implementation

The IKE daemon appears to be a modified variant of the raccoon2 project, compiled with the GNU MP library. Unlike the TLS implementation, all randomness used by the daemon is output by `/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 output from the PRNG to recover the state.

In the IKEv1 implementation, the first block of RNG output is used to generate the IKEv1 cookie by hashing it together with IP addresses, ports, and memory addresses. The cookie was generated as

$\texttt{SHA1(0x2020||mpz\_d||src||dst||timestamp||nonce}_{16}\texttt{)}$.[7] 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.

In the IKEv2 implementation, the SPI field, the equivalent of the IKEv1 cookie, is eight raw bytes of PRNG output.

In both IKEv1 and IKEv2, the next block of RNG output was used to generate the handshake nonce, which was 16 bytes long. This was generated immediately before the RNG 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. The cryptographic processor deterministically transforms the exponent in a way that we did not manage 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.

F. State recovery in IKEv1

The state recovery attack outlined in Section II-C requires two blocks of RNG 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 a second, so we assume it is known. However, the heap-allocated pointer provides approximately 13 bits of entropy [30]. Rearranging Equation 2, the first block of RNG 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 RNG 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 entirely feasible. We found that IKEv2 state recovery was more easily exploitable, and focused our efforts on IKEv2 as described below.

The two blocks after the cookie and nonce are used to generate the Diffie-Hellman key private key, so key recovery would be straightforward after recovering the state.

G. State recovery in IKEv2

As discussed in Section II-E2, we need two consecutive blocks of the RNG output, an approximation to the two timestamps used for the intermediate vectors, and the AES key in order to recover the state. The FortiOS IKEv2 implementation yields 1.5 consecutive blocks of raw RNG 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 II-C to recover the RNG state.

The difference between the receive time of the first handshake packet received and the timestamp generated by `gettimeofday()` in the RNG when it was called to generate the SPI cookie is dependent on the time taken by the Fortigate device's processor 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. We found that searching within a ten-second window, or about 2^{24} guesses for the first timestamp, worked well on our hardware as well as scanned machines in the wild.

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

This yields a total state recovery attack complexity on similar hardware of about $2^{24} \cdot 2^5 = 2^{29}$ timestamp guesses, given complete uncertainty within 10 seconds for the first block (2^{24} guesses), and bounding the search space for the second timestamp to within 3σ of the mean (2^5 guesses).

Experimentally, this can be completed for both 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 V, 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, at most $2^{29}/2^{64} = 2^{-35}$.

H. 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 only verifying Equation 2 with 1.75 blocks of raw output, the reduced size of the timestamp search means that the probability of a false positive match is small. For a timestamp search space of 2^{29} (see Figure 4), the expected number of false positive matches is at most $2^{29}/2^{96} = 2^{-67}$.

I. 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, simultaneously guessing the two

[7] We note the choice of SHA1 here over MD5 recommended in the RFC.

following timestamps in order to recover two more blocks R_2 and R_3 that will be used to generate the Diffie-Hellman secret. To obtain the first key block R_2, we apply Equation 3 and then wind forward the generator, again guessing the next timestamp as in Equation 1 to generate R_3.

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. A match confirms our guesses for the two timestamps and the Diffie-Hellman private value, that can then be used to calculate the shared secret.

We measured the time difference between the nonce PRG timestamp and the first key block PRG timestamp and found a mean of 154.4μs with a standard deviation of 32.2μs. We search 3σ out from the average to find this timestamp, requiring a search over 2^8 timestamps. We also measured the average difference between the first and second calls to the PRG at 18.3μs with a standard deviation of 4.53μs for the Fortigate 100D and 1141μs for the virtual appliance over 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 $2^8 \cdot 2^5 = 2^{12}$ 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 RNG outputs. Over 30 trials, the average time to carry out this part of the attack was 3.88s on the hardware device itself.

J. 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 II-E2 and verified full passive decryption against traffic to our FortiGate 100D.

V. Measurements

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

TABLE II: 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	23,554
...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

A. HTTPS

We used several types of HTTP and HTTPS metadata to identify affected Fortigate hosts in the wild. Our scans targeted hosts that exposed the administration panel for the device on a public IPv4 address on port 443.

a) 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 TLS/SSL handshake. Our scan connected over TLSv1.0, the version supported by the vulnerable devices, and offered the same list of cipher suites that our test device supported. We give the list of cipher suites supported by FortiOSv4 and advertised in our scans in Table III in Appendix B. Our scan successfully negotiated a HTTPS connection with 29,709,242 hosts. (This is lower than the roughly 40 million HTTPS hosts seen in Internet-wide scans that offer a wider variety of SSL/TLS versions and cipher suites.)

b) Server certificate common name: The default configuration for FortiOS v4 is to serve a self-signed certificate with the model number and serial number in the common name field and 'Fortinet' in the organizational field. While this does not identify the firmware or build number for a host, it provided a subset of hosts to test further for the vulnerability. We found 114,172 hosts in our scan whose certificate contained a matching organization field. The certificate common names for these hosts listed 3,379 unique model numbers.

We were able to successfully mount the state recovery attack against 23,554 of these hosts, corresponding to 20.6% of those with default Fortinet certificates. We initially searched a 10 second window around the local timestamp of the first packet arrival time, and were able to recover state for 16,418 hosts. We were able to perform state recovery for an additional 7,136 hosts by searching out from the timestamp in the received server random.

Figure 4 shows the distribution of the number of timestamps guessed for successful state recoveries. We searched a 100μs window between successive timestamps in these trials to account for any hardware that may have been slower than the mid-range FortiGate 100D we tested with, but did not observe any differences beyond 35μs among our successful state recoveries. We plot the distribution of gaps between timestamps in Figure 6.

c) Specific HTTP files: The the administration panel of our hardware device contained an image file located at `/images/logon.gif`. In our HTTPS scan, after success-

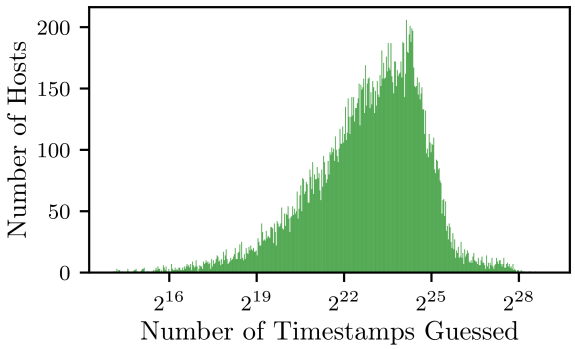

Fig. 4: **Brute force work for state recovery.** We plot the distribution of the number of timestamps we were required to guess for each successful TLS state recovery. The average number of guesses we required was $2^{24.4}$ with a standard deviation of $2^{23.8}$.

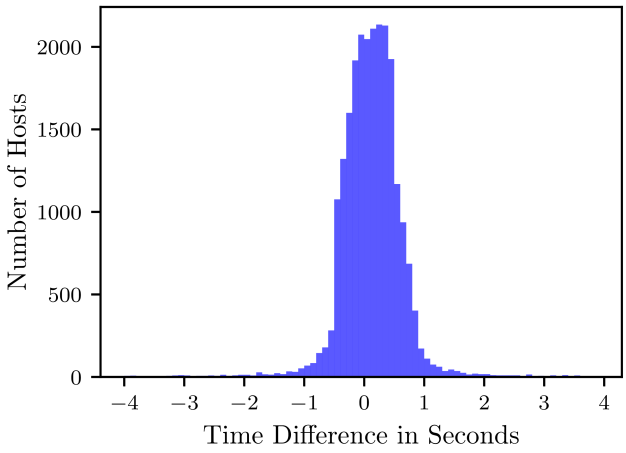

Fig. 5: **Initial timestamp offset.** We calculated the difference between the time a packet containing the TLS Server Hello was received and the timestamp used to seed the first PRNG block for the packet from successful state recovery trials in the wild. We brute force searched for this timestamp within a 10-second window of our local timestamp. If that was unsuccessful then we also searched from the time indicated in the server random. The average difference is 0.11 seconds with a standard deviation of 0.59 seconds.

fully negotiating the HTTPS handshake, we then sent HTTP GET request for this URL. 605,950 hosts responded with HTTP OK, and a corresponding image. (The others returned a 404 error). We were unable to automatically validate which images corresponded to Fortigate devices based on image data alone, thus we used the below matching techniques to further narrow candidates.

d) ETag headers: The HTTP ETag header gives a value associated with resources on an HTTP server, and used for web cache validation along with conditional requests [31]. The RFC specifies that the value of the header "is data known only to the server". In order to attempt to fingerprint devices running vulnerable firmware versions, we matched ETag headers obtained from requests to the responding hosts against a list of ETags corresponding to FortiOS v4 devices.

The Equation Group leak [32] contained a list of pairings between ETag suffixes and device and firmware-build pairs. The leaked list is far from complete, but 168 entries out of the total 440 corresponded to 9 models running 26 different builds of FortiOS v4. The leak additionally 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 an ETag identified as FortiOS v4. The state recovery attack was successful for 2,265 (97%) of these. 1,535 of the hosts with matching ETags presented HTTPS certificates other than the default Fortigate certificate. State recovery was successful for all the devices matching the ETag for our hardware device.

e) Limitations: We note that devices responding to our HTTPS scans have been configured to expose the administration panel on a public IPv4 address, which is not the default configuration and increases the vulnerability of the devices. (On our test hardware, the admin panel was not exposed at all by default, and we had the option to configure an interface to expose it on setup.) The total population of vulnerable hosts is likely higher than the population visible to our scan.

A well equipped adversary could construct a larger database of ETags and by doing so enable further firmware version fingerprinting from the large set of hosts. This technique can also be used for other manufactures, a number of whom also insert a model-firmware identifier in header.

B. IKEv2

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

a) 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 listed a variety of Fortinet model numbers. From the remaining 17 hosts whose nonces and key exchanges were freshly generated on new connections, our key recovery attack was successful against 7.

b) Cipher support: In our scans, we sent handshake requests with proposals requesting the modp Diffie-Hellman groups from sizes 768 to 2048, encryption algorithms of AES, 3DES and DES, SHA-1 or SHA-2 for the hash algorithm and either preshared key or RSA authentication. These proposals were chosen because they were supported in FortiOS v4. We give the list of cipher suites we offered in our scan in Table V in Appendix B. We received 7,743,876 responses.

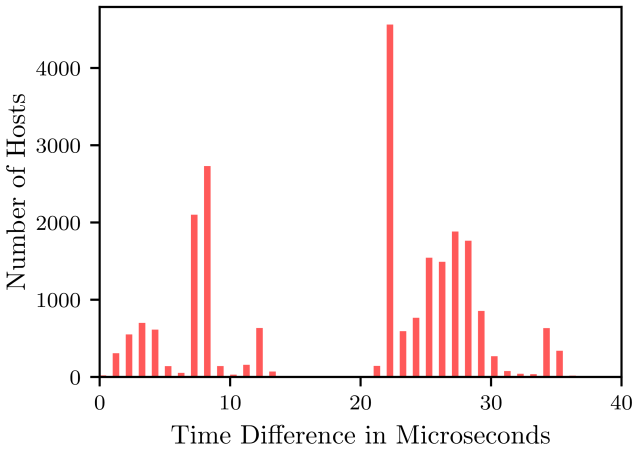

Fig. 6: **Subsesquent timestamp offset.** We calculated the difference between the first and second timestamps used to generate the RNG blocks for the TLS nonce. This value was brute forced from within a range of between zero and one hundred microseconds. The average difference is 19.2 microseconds with a standard deviation of 10.1 microseconds.

c) 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.

d) 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.

C. Timestamp Statistics

Our state recovery attack was configured to examine a window of ten seconds around the local timestamp of when we received the packets during our scan. In Figure 5, we plot the distribution of the first recovered timestamp relative to the local timestamp at the time we received the first packet for 24,000 successful state recovery attacks against TLS nonces in the wild. In nearly all cases, the first timestamp recorded in the RNG state was less than one second away from the local timestamp of our scanning machine at the time the packet was received. In Figure 6, we plot the difference between the first and second timestamps used to generate the TLS nonce; 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.

VI. Related Work

A. 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 [8] 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 [33] analyzed the Linux random number generator in 2006, and Dorrendorf, Gutterman, and Pinkas [34] analyzed the Windows random number generator in 2009. Dodis et al. [9] 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 [35] analyzed Java random number generation implementations and noted several vulnerabilities, including a vulnerability in Android.

B. Random number generation failures

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

One category of RNG failures in cryptographic practice appear 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. [2] 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 [24]. Lenstra et al. [36] 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 [37] performed a follow-up study that found low to nonexistent software patching rates for systems affected by the 2012 RNG flaws. Bernstein et al. [38] 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. [39] performed a similar analysis on the 3.7 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 [40] 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 [41].

C. Intentional RNG backdoors

A further category of failures are due to designs with intentional weaknesses. Young and Yung [42] introduced the concept of kleptography, the design of cryptosgraphic chemes with hidden backdoors. They later described a scheme for introducing such a backdoor into discrete log problem based cryptosystems [43].

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 [44] noted that the indistinguishability proof for the NIST-standardized Dual EC DRBG relies on a random Q parameter. Shumow and Ferguson [45] 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. [3] 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. [15] formally model backdoored random number generators, design backdoored PRNGs with strong indistinguishability properties, and evaluate countermeasures against backdoors.

VII. Discussion

A. NSA decryption capabilities

Classified NSA documents leaked by Edward Snowden and published by Der Spiegel [46] 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 [47] (Boudot [48] 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 [16], [3], and software exploits and malware ("implants").

We suspect the reality is a combination of these techniques customized to individual 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.

B. 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 [3] 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 in the nonce length.

C. 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, physical access, or similar. 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.

D. 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.

E. Concerns about other PRG implementations

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 cryptographic implementations are sound. Cipher-based PRGs appear specifically vulnerable to state recovery attacks when the cipher key is known or obtained by an attacker. This raises the possibility that a careless or malicious implementation of a modern PRG such as NIST's CTR_DRBG [5] could

be implemented in such a way that the key is *not* routinely updated, which might allow for 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.

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, and EFMA-1441209; The Office of Naval Research under contract N00014-14-1-0333; 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] P. Q. Nguyen and I. E. Shparlinski, "The insecurity of the Elliptic curve Digital Signature Algorithm with partially known nonces," *Designs, codes and cryptography*, vol. 30, no. 2, pp. 201–217, 2003.

[2] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: Results from the 2008 Debian OpenSSL vulnerability," in *Proceedings of IMC 2009*, A. Feldmann and L. Mathy, Eds. ACM Press, Nov. 2009, pp. 15–27.

[3] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the Juniper Dual EC incident," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 468–479.

[4] N. Perlroth, "Government Announces Steps to Restore Confidence on Encryption Standards," *The New York Times*, 2013.

[5] E. B. Barker and J. M. Kelsey, *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, 2007.

[6] J. Kelsey, B. Schneier, and N. Ferguson, "Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator," in *SAC '99*, 1999.

[7] S. Müller, "Linux random number generator — a new approach," Available at http://www.chronox.de/lrng/doc/lrng.html.

[8] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic attacks on pseudorandom number generators," in *Fast Software Encryption*. Springer, 1998, pp. 168–188.

[9] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs, "Security analysis of pseudo-random number generators with input: /dev/random is not robust," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 647–658.

[10] F. Strenzke, "An analysis of OpenSSL's random number generator," in *EUROCRYPT '16*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 644–669. [Online]. Available: https://doi.org/10.1007/978-3-662-49890-3_25

[11] S. Ruhault, "SoK: Security Models for Pseudo-Random Number Generators," in *IACR Transactions on Symmetric Cryptography (TOSC)*, vol. 1, 2017.

[12] Y. Dodis, A. Shamir, N. Stephens-Davidowitz, and D. Wichs, "How to eat your entropy and have it too – optimal recovery strategies for compromised RNGs," in *CRYPTO '14*, 2014.

[13] S. S. Keller, "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, 2005.

[14] S. S. K. Lawrence E. Bassham III, "The Random Number Generator Validation System (RNGVS)," National Institute of Standards and Technology, 2005.

[15] Y. Dodis, C. Ganesh, A. Golovnev, A. Juels, and T. Ristenpart, *A Formal Treatment of Backdoored Pseudorandom Generators*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 101–126. [Online]. Available: https://doi.org/10.1007/978-3-662-46800-5_5

[16] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the Juniper Dual EC incident," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 468–479.

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

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

[19] E. C. Kaufman, "Internet Key Exchange (IKEv2) protocol," IETF RFC RFC4306, 2005.

[20] NIST, "Cmvp historical validation list," http://web.archive.org/web/20170120035228/http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-historical.htm, January 2017.

[21] ——, "Security requirements for cryptographic modules," http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf, May 2001.

[22] E. Barker and A. Roginsky, "Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths," *NIST Special Publication*, vol. 800, p. 131A, 2011.

[23] "FIPS 140-2 SECURITY POLICY FOR: INZERO GATEWAY."

[24] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," in *Proceedings of USENIX Security 2012*, T. Kohno, Ed. USENIX, Aug. 2012.

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

[26] MITRE Corporation, "CVE-2016-8492." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8492

[27] M. Corporation, "Cve-2016-1909." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1909

[28] Fortinet, "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, January 2016.

[29] W. J. van der Laan, "Tadaqueous moments," Sep 2016. [Online]. Available: http://laanwj.github.io/2016/09/01/tadaqueos.html

[30] W. Herlands, T. Hobson, and P. Donovan, "Effective Entropy for Memory Randomization Defenses," in *USENIX 7th Workshop on Cyber Security Experimentation and Test*. Lincoln Labratory, Aug. 2014.

[31] J. Franks, P. M. Hallam-Baker, J. L. Hostetler, S. D. Lawrence, P. J. Leach, A. Luotonen, and L. C. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," Internet Requests for Comments, RFC Editor, RFC 2617, June 1999, http://www.rfc-editor.org/rfc/rfc2617.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2617.txt

[32] D. Goodin, "Group claims to hack NSA-tied hackers, posts exploits as proof," Aug 2016. [Online]. Available: https://arstechnica.com/information-technology/2016/08/group-claims-to-hack-nsa-tied-hackers-posts-exploits-as-proof/

[33] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the Linux random number generator," in *IEEE Symposium on Security and Privacy*. IEEE Press, 2006.

[34] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the random number generator of the Windows operating system," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 10, 2009.

[35] K. Michaelis, C. Meyer, and J. Schwenk, "Randomly Failed! The State of Randomness in Current Java Implementations." in *CT-RSA*, vol. 7779. Springer, 2013, pp. 129–144.

[36] A. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, Whit is right," IACR, Tech. Rep., 2012.

[37] M. Hastings, J. Fried, and N. Heninger, "Weak keys remain widespread in network devices," in *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM, 2016, pp. 49–63.

[38] D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. Van Someren, "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, 2013, pp. 341–360.

[39] G. Kadianakis, C. V. Roberts, L. M. Roberts, and P. Winter, ""Major key alert!" Anomalous keys in Tor relays."

[40] T. Ristenpart and S. Yilek, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography," in *NDSS '10*, 2010.

[41] A. Klyubin, "Some securerandom thoughts," https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html, August 2013.

[42] A. Young and M. Yung, "Kleptography: Using cryptography against cryptography," in *Eurocrypt*, vol. 97. Springer, 1997, pp. 62–74.

[43] ——, "The prevalence of kleptographic attacks on discrete-log based cryptosystems," in *Annual International Cryptology Conference*. Springer, 1997, pp. 264–276.

[44] D. R. Brown, "Conjectured security of the ANSI-NIST elliptic curve RNG," *IACR Cryptology ePrint Archive*, vol. 2006, p. 117, 2006.

[45] D. Shumow and N. Ferguson, "On the possibility of a Back Door in the NIST SP800-90 Dual EC PRNG." [Online]. Available: http://rump2007.cr.yp.to/15-shumow.pdf

[46] "Intro to the VPN exploitation process," Media leak, Sep. 2010, http://www.spiegel.de/media/media-35515.pdf.

[47] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[48] F. Boudot, "On improving integer factorization and discrete logarithm computation using partial triangulation," Cryptology ePrint Archive, Report 2017/758, 2017, http://eprint.iacr.org/2017/758.

APPENDIX A
THE FORTIOS V 4 X9.31 INITIALIZATION ROUTINE

Listing 1: The X9.31 Initialization Routine.

```
1  int initialize_X931()
2  {
3    char rng_state[16];
4    char timestamp_buffer[16];
5    int aes_key[4];
6    int result = key_set;
7    aes_key[0] = 0x6D66B1F3;
8    aes_key[1] = 0x42726013;
9    aes_key[2] = 0xAB1C06ED;
10   aes_key[3] = 0x0262D4B8;
11   if ( !key_set )
12     result = set_aeskey(aes_key);
13   if ( !state_set )
14   {
15      // initial state setting removed for clarity
16         ...
17     save_state(rng_state);
18     fill_timestamp(timestamp_buffer);
19     result = x931(&timestamp_buffer, output_buffer,
20                   rng_state, 16);
21   }
22   return result;
23 }
```

APPENDIX B
SUPPORTED CIPHER SUITES IN FORTIOSV4

TABLE III: 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 IV: 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 V: 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			