TOO IMPORTANT TO LEAVE TO CHANCE: PSEUDORANDOM NUMBER GENERATOR STANDARDIZATION & SECURITY

Shaanan Natanel Cohney

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2019

Supervisor of Dissertation

Nadia Heninger, Adjunct Associate Professor of Computer and Information Science

Co-Supervisor of Dissertation

Jonathan M. Smith, Professor of Computer and Information Science

Graduate Group Chairperson

Rajeev Alur, Computer and Information Science Graduate Group Chair

Dissertation Committee

Christopher S. Yoo
Professor of Law, Communication, and Computer and Information Science

Matthew A. Blaze
Adjunct Professor of Computer and Information Science

Matthew D. Green
Associate Professor, Johns Hopkins Information Security Institute

Boon Thau Loo
Professor of Computer and Information Science

TOO IMPORTANT TO LEAVE TO CHANCE: PSEUDORANDOM NUMBER GENERATOR STANDARDIZATION & SECURITY

© COPYRIGHT

2019

Shaanan Natanel Cohney

This work is licensed under the
Creative Commons Attribution
NonCommercial-ShareAlike 3.0
License

To view a copy of this license, visit

http://creativecommons.org/licenses/by-nc-sa/3.0/

Dedicated to my grandparents—Saba, Savta, Nanna, and Shimmy

ACKNOWLEDGMENT

I have been immensely lucky to have more than my fair share of mentors, friends, family, co-conspirators, and teachers over the years leading to this work.

Nadia Heninger, for the adventures, the long nights of guidance, research and snacks, and for the Phillip Glass soundtrack to my PhD. JMS for teaching me not to jump in puddles, and for bringing me in 'off the street'. Matt Blaze for encouraging me to bend rules and expectations, and for saving my bacon when things went wrong. Boon Thau Loo for the decade he put into my education, and for his enthusiasm each step of the way. Matt Green for the supportive tweets and tweaks, and brilliant ideas. Professor Christopher S. Yoo, for guiding my journey from misguided, to legally-educated, engineer.

A few exceptional teachers and professors stand out for making this work possible. Dr. Val Tannen, Dr. Rachel 'Doc' Forgasz, and Alan Madden for fundamentally shaping my academic journey. Dr. Steven Prawer and Dr. Elisabetta Barberio for my first taste of research. Professor David Hoffman for the (de)motivating tweets, and Chris Soghoian for teaching me to write.

My friends down-under never failed to welcome me back again with each visit, and that gave me comfort being away so long. Those who provided particular support over the years include Tomer Simovich, Amanda Horiniak, E.B Zimet, Lenny Bloom, Mark Bloom, Michael Erlichster, Tessa Satherly, Joyce Kant, Ariel Zeleznikow-Johnston and Esther Cochrane Davis. Likewise, this would not have been possible without the support of those closer to Philadelphia: Aviva Rosen, Shira Silver, Dagean Golomb, Alec Webley, Alexis Block, Rocky Neumann, Katie Gardner, Sarah Silverberg, Sharon Christner, Bhavish Malkani, Stephanie Li, Arianna Unger, Alicia Lai, and Daniel Carsello. My time would not have been complete without the members of the Penn Glee Club and the Philomathean Society with whom my non-working hours were spent.

The members of the Security Group at Penn deserve special thanks for their ongoing companionship and support. I wish to thank Dr. Sandy Clark, Bill Cheswick and Robert Broberg for the bright sparks and long conversations on everything security. Josh Fried, for both the mess we made, and the way we

cleaned it up. Marcella Hastings for being a constant ear for me, aide in writing, for hanging on with me when the lab got quiet, and for being a sounding board for every little complaint. Gabrielle De Micheli for the dinners, talks, tears, and emotional support you provided me. For the constant check-ins, and the understanding you always showed. Luke Valenta, my 'lawful good' partner-in-crime, dear friend, co-author, and co-conspirator – whenever I felt like I was falling behind, you were always there to set goals with me, sometimes two steps ahead and sometimes one behind, but at the end, always by my side.

I owe a special debt of gratitude to my second-family: Riepe College House and the faculty who adopted me so readily. Dr. Betsy, Anya and Charlie Rymes for the pancakes. Dr. Ralph and Ellen Rosen for the constant listening ears, Dr. Dennis DeTurk, for the hundreds of hours of piano and the hundreds of thousands of cookies, Dr. Amy and Vincenzo Stornaiuolo, for the support writing and pasta, Dr. Sebastián Gil-Riano and Adriann Moss for the ice cream and insight into Australian academia. I am grateful to the many hundreds of residents and RAGAs who shared their love with me, among them Selinda Qiu, Laura Whelan, Natasha Napolitano, Nadia Park, Daniel Brennan, Sonia Gandhi, Maria Kiameso DaSilva, Kaitlyn Lapp, and Andrea Klein. Dean Marilynne Diggs-Thompson, I can't describe how grateful I am for the warmth (and food) you provided me and for the constant attention you paid to my wellbeing.

To Yoni, Rifki, Tzippi and Sophie for being my actual family away from home. My grandparents Saba, Savta, Nanna and Shimmy, for the everlasting and unconditional support. To my brothers, Adiel and Raphael, for their constant enthusiasm, and interest in my life and for always being there for me. Finally, to my parents, Shelley and Shlomo, who called nearly every day since I left home and who have been an immeasurable well of support and inspiration.

ABSTRACT

TOO IMPORTANT TO LEAVE TO CHANCE: PSEUDORANDOM NUMBER GENERATOR STANDARDIZATION & SECURITY

Shaanan Natanel Cohney

Nadia Heninger

Jonathan M. Smith

This dissertation addresses the security of pseudorandom number generators (PRGs), illustrating that flaws persist within key standards despite the purported effectiveness of standardization and certification processes. By evaluating three standardized designs and developing real-world attacks against each, I show how an adversary who is able to introduce flaws into a standard can compromise real implementations. Such 'pre-supply chain operations' (PSYCHOs) are within the capabilities of state actors who, as I evidence, may have already incorporated similar attacks into their strategic portfolios.

My first case study is an attack on the ANSI X9.31 PRG highlighting the vulnerability of standards in the absence of correct advice concerning keying and key rotation. The analysis illustrates how a known flaw in a design can persist through multiple rounds of review, and ultimately impact certified devices. I demonstrate that the presence of this flaw in one particular manufacturer's devices enables a passive traffic decryption exploit against the IKEv2 virtual private network (VPN) protocol.

The next case study examines the Dual_EC PRG design which contains a potential backdoor, contingent on how certain parameters in the standard were chosen. Unlike the flaw in X9.31, credible reporting indicates that the flaw was an intentional design outcome and the result of intervention by U.S. intelligence agencies. I build a reconstructed timeline of how the flaw impacted the Juniper ScreenOS line of devices, along with a demonstration of how it can be used to achieve passive VPN decryption.

The final study develops attacks on the CTR_DRBG design, which rely on the absence of clear guidance on reseeding and exclusion of side channel attacks from the corresponding standard's threat model. Using

these attacks I show how an adversary can compromise long term TLS authentication keys belonging to a client using the design.

I conclude by drawing together these threads to taxonomize different PSYCHO attack vectors which are potentially attractive for highly sophisticated adversaries with long time horizons and institutional influence.

TABLE OF CONTENTS

ACKNOWLEDGMENT iv

ABSTRACT vi

TABLE OF CONTENTS viii

LIST OF TABLES xi

LIST OF ILLUSTRATIONS xii

CONTRIBUTING PUBLICATIONS xiii

CHAPTER 1 : Introduction 1

1.1 Motivation 4

1.2 Research Questions & Contributions 5

1.3 Methodologies 6

1.4 Overview 7

CHAPTER 2 : Background 9

2.1 Formalism 10

2.2 FIPS and FISMA 12

2.3 Prior Work 15

CHAPTER 3 : Early Signs of Trouble 19

3.1 Introduction 21

3.2 Background 23

3.3 FIPS and Hard-coded X9.31 Keys 30

3.4 Decrypting VPN traffic on FortiOS v4.3 34

3.5 Measurements 43

CHAPTER 4 : The Evil Inclination 49
4.1 Introduction 51
4.2 Dual EC 53
4.3 History of the Juniper Incident 55
4.4 The ScreenOS PRG 58
4.5 Interaction with IKE 61
4.6 Attacking IKE 69
4.7 Passively detecting ScreenOS 71
4.8 Counting ScreenOS Devices 72
4.9 Discussion 75
4.10 Related Work on Dual EC 80

CHAPTER 5 : Silence on Side-channels 81
5.1 Introduction 83
5.2 Background 87
5.3 CTR_DRBG 90
5.4 Cryptanalysis of CTR_DRBG 94
5.5 State Recovery Attack 95
5.6 Cache Attack Details 97
5.7 Attacking TLS 103
5.8 Attacking Full Entropy Implementations 114
5.9 Impact 123
5.10 Discussion 124

CHAPTER 6 : From the Ashes 125
6.1 Comparing Attacks 126
6.2 Lessons From PRGs 128
6.3 Generating Random Numbers Today 129
6.4 Standardizing PRGs 131

CHAPTER 7 : PSYCHOs 133
7.1 Snowden Redux 135
7.2 Expanding Threat Models 136
7.3 Dangerous Defaults 141
7.4 A Failed PSYCHO? 146
7.5 Contextualizing the Attacks 149
7.6 A Better Tomorrow 151

CHAPTER A : Ethics & Disclosure 157

BIBLIOGRAPHY 160

LIST OF TABLES

TABLE 1 : Number of CMVP Certificates Issued to PRGs 15

TABLE 2 : Certificate Issuances for X9.31 31

TABLE 3 : X9.31 Misuse in CMVP Databse 31

TABLE 4 : Affected Implementation Versions 34

TABLE 5 : Scanning Parameters Used to Find FortiOS Devices. 44

TABLE 6 : X9.31 State and Key Recovery in the Wild 48

TABLE 7 : ScreenOS firmware versions we examined 59

TABLE 8 : ScreenOS Features by Version 77

TABLE 9 : Nonce Recovery Complexity 113

TABLE 10 : CMVP-Certified Uses of DRBG Designs 123

TABLE 11 : CTR_DRBG Use in CMVP Database 124

TABLE 12 : Comparison of Analyzed Attacks 127

TABLE 13 : Comparison of Possible PSYCHOs 150

LIST OF ILLUSTRATIONS

FIGURE 1 : ANSI X9.31 PRG 25

FIGURE 2 : Randomness and the IKEv2 Handshake 28

FIGURE 3 : Counting vulnerable implementations 33

FIGURE 4 : Brute force work for TLS state recovery 45

FIGURE 5 : Recovering Timestamps from TLS Handshakes 46

FIGURE 6 : Nonce Queue Behavior 65

FIGURE 7 : Nonce Queue State Following Exchanges 65

FIGURE 8 : CTR_DRBG Output Loop 91

FIGURE 9 : Probe Alignment with Encryptions 100

FIGURE 10 : Key Recovery Rate with Prefetcher Enabled 102

FIGURE 11 : PKCS#1v1.5 RSA Padding Structure 104

FIGURE 12 : Single byte differential propagation in AES state 118

FIGURE 13 : Clay Tablet with inscription of Annals of Tiglath-Pileser III 136

FIGURE 14 : MITRE Corp. Supply Chain Attack Model 137

FIGURE 15 : Comparing Supply Chain Attacks and PSYCHOs 140

CONTRIBUTING PUBLICATIONS

Primary Works

1. Shaanan Cohney, Andrew Kwong, Shachar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, Yuval Yarom. "Pseudorandom Black Swans: Cache Attacks on CTR_DRBG". In: *IEEE Security and Privacy, 2020.*
2. Shaanan Cohney, Nadia Heninger, Matthew D. Green. "Practical state recovery attacks against legacy RNG implementations". In: *ACM Conference on Computer and Communications Security (CCS), 2018*
3. Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, Hovav Shacham. "A Systematic Analysis of the Juniper Dual EC Incident". In: *ACM Conference on Computer and Communications Security (CCS), 2016.*

Secondary Works

1. Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, Nadia Heninger. "Measuring small subgroup attacks against Diffie-Hellman". *In The Network and Distributed System Security Symposium 2017.*
2. Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. "DROWN: Breaking TLS using SSLv2". In: *25th USENIX Security Symposium, 2016.*

CHAPTER 1

INTRODUCTION

Guide of our youth; Lead thou thy children on; To light and truth;

Edgar M. Dilley

It is a truth universally acknowledged, that a securely implemented cryptographic primitive must be in want of a cryptographically secure pseudorandom number generator [1]. Modern cryptography relies on randomness to prevent an attacker from predicting secret values generated by parties in a cryptographic protocol. Indeed, random values are universally used to ensure security properties for nearly all cryptographic data, including secret keys for confidentiality or integrity, secret keys for public-key encryption, key exchange, or signatures, as well as for protocol nonces to prevent replay attacks.

Unfortunately, high bandwidth, high quality sources of randomness have not historically been available to most digital devices. While designs for mechanical production of random digits have existed at least

[1] Jane Austen. *Pride and Prejudice*. 1813.

since 1947 [2], the widespread integration of True Random Number Generators (TRNGs) into modern processors is a more recent phenomenon [3,4].

Further, while both hardware and mechanical sources do provide a measure of true 'randomness' they still exhibit biases in their output, which necessitates further processing to whiten the outputs.

The need for software based random number generators to perform this whitening, and the need for random numbers in contexts where TRNGs are not available, both motivate the design and use of pseudorandom number generators (PRGs). Thus, a cryptographically secure Pseudorandom Generator (PRG) is one of the fundamental primitives of modern cryptography, both in theory and in practice.

The simplest theoretical PRG construction is an algorithm that expands a smaller seed into a longer output sequence that is computationally indistinguishable from a true sequence of random bits. However, the practical security demands for random number generation are somewhat more complex; in real systems, these pseudorandom number generator constructions are often multi-stage algorithms that collect inputs from environmental entropy sources or hardware into an "entropy pool". The pool is then used to seed a PRG that generates cryptographically secure output. Real world PRGs must also meet additional security guarantees, including recovery from state compromise.

The problem of constructing random and pseudorandom number generators has been extensively explored within industry [5–7] and in the academic literature [8–13]. Despite the abundant results of this effort industry has consistently relied on a small number of common pseudorandom number generation

[2] George W. Brown. *History of RAND's Random Digits*. Tech. rep. P-113. 1949

[3] John M. *The Difference Between RDRAND and RDSEED*. Online: https://software.intel.com/en-us/blogs/2012/11/17/the-difference-between-rdrand-and-rdseed. Intel Corporation, 2012

[4] Advanced Micro Devices (AMD). *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Tech. rep. 2015

[5] Elaine Barker and John Kelsey. *NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology, 2007. 109

[6] John Kelsey, Bruce Schneier, and Neils Ferguson. *SAC*, 1999.

[7] Stephan Müller. *Linux Random Number Generator — A New Approach*. Online: http://www.chronox.de/lrng/doc/lrng.html

[8] John Kelsey et al. *FSE*, 1998.

[9] Yevgeniy Dodis et al. *ACM CCS*, 2013.

[10] Falko Strenzke. *EUROCRYPT 2016*, 2016.

[11] Sylvain Ruhault. *FSE*, 2017.

[12] Yevgeniy Dodis et al. *CRYPTO*, 2014.

[13] Sandro Coretti et al. *CRYPTO*, 2019.

algorithms. This may be attributable in part to implementers desire to adhere to U.S. cryptographic standards, which opens up access to government contracts. These standards allow use of a limited number of designs. For example, until 2007 there were only *two* algorithms for pseudorandom number generation approved for U.S. government uses, 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.

PRG Standards Public government standards for cryptography have existed since 1975, when the National Bureau of Standards (now NIST, the National Institute of Standards and Technology) published the Data Encryption Standard (DES) [14]. In the following years, as the need for interoperability, security and confidence in computer systems grew, such standards proliferated. Particularly prevalence of high-risk computer systems both within government, and in regulated industries such as health-care and business, necessitated a governmental strategy for ensuring the security of data both at rest, and in transit.

The first federal standards for PRG design arose in the mid-1990s, beginning with the publication of the Federal Information Processing Standard (FIPS) 171, which permitted use of a design described in another standard, ANSI X9.17. As PRG standards matured, NIST, the agency responsible for standardization, instituted verification and certification programs to allow manufacturers and implementers to ensure their products conformed to the set standards. The modern incarnation of the certification program—the Cryptographic Module Validation Program (CMVP)—outsourced testing to third party labs. For validation, NIST publishes sets of test vectors for each approved algorithm, against which an implementer can check the outputs from their system.

The existence of federally endorsed minimum security standards for cryptography influenced vendors to include, and recommend use of their products in conformance with the standard [15]. Further, these vendors have at times advertised compliance to FIPS as a security feature. This is indicative that not only did vendors historically believe that adherence to the FIPS standard ensured a higher level of effective

[14] U.S. Congress. *Defending Secrets, Sharing Data: New Locks and Keys for Electronic Information: Appendix C*. tech. rep. OTA-CIT-310. Office of Technology Assessment, Oct. 1987

[15] Aaron Margosis. *Why We're Not Recommending "FIPS Mode" Anymore*. Online: https://blogs.technet.microsoft.com/secguide/2014/04/07/why-were-not-recommending-fips-mode-anymore/. Microsoft, 2014

security, but that end-users are motivated by the allure of a seal-of-approval style 'solution' to security problems.

1.1 Motivation

The real world impacts of PRG design failure combined with the lack of diversity in available implementations impels a closer look at the security of these designs. In light of media reports and leaks of classified material [16,17] suggesting that the U.S. government may have compromised standards, the strong tie between standardization and industry adoption motivates a re-evaluation as to whether the current standardization regime is helping or harming efforts to ensure cryptosystem security.

Mismatches Between Theory and Practice. Significant effort has been dedicated to formalizing PRG security properties and designing provably secure constructions. However, theoretical analyses of many of the most commonly-used designs in practice (the Linux RNG [9], CTR_DRBG [18]) have found that these designs do not meet basic security properties such as robustness against state compromise. Unfortunately, implementers are often hesitant to adopt countermeasures without a concrete demonstration of vulnerability. This motivates the applied aspect of this work and drives the need for measurement and demonstration of real world exploits.

[16] National Security Agency. *Intro to the VPN Exploitation Process*. Media leak. Sept. 2010
[17] Nicole Perlroth. *Government Announces Steps to Restore Confidence on Encryption Standards*. 2013
[18] Joanne Woodage and Dan Shumow. *EUROCRYPT*, 2019.

1.2 Research Questions & Contributions

Research Questions

- Are the FIPS certified pseudorandom number generators secure as deployed in real world settings? To what measure are their security properties explained by both the standards, and by the standardization and certification processes?
- To what extent does an adversarial model incorporating the standardization process cohere with known attacks on real world systems?



Contributions

This work shows that despite the best efforts of standardizing bodies, FIPS compliance has at times not only failed to improve product security but has increased the likelihood that a product contains vulnerabilities that would have been excluded were the manufacturer following cryptographic best practices.

My broader contribution develops the notion of *pre-supply chain operations* (PSYCHOs), whereby a powerful attacker with institutional influence uses standardization, design and certification processes to introduce exploitable flaws into deployed cryptosystems.

1.3 Methodologies

(A) Evaluation of PRGs. To evaluate the impact of standardization on real-world PRG security, I systematically evaluate the set of U.S. government standardized PRGs. As detailed in Chapter 6, there is significant prior work evaluating the theoretical properties of these generators, some of which includes descriptions of where existing security proofs are incomplete or lacking. I identify potential vulnerabilities in real world PRGs by reading the standards, theoretical work, and manufacturer provided documentation as an integrated whole. The technique of exhaustively surveying CMVP (or other similar) certification documents to identify vulnerabilities is a novel methodology, and a contribution in and of itself. Once I identify products that may contain vulnerabilities, I reverse engineer the PRG implementations within to discern if they are deployed in a vulnerable fashion. If so, I demonstrate how the product can be exploited in a natural configuration, under realistic assumptions.

(B) Building a Threat Model. The attacks developed Chapters 3–5 are the product of a diverse set of standardization failure modes. I expand my scope by assessing instances of standards failure identified by other authors, in addition to a secondary work to which I contributed. By categorizing these failure modes, I develop a taxonomy and build a threat model that incorporates standardization-time risks. While the model is retrospective by necessity, it serves as an aid in hardening future design and standardization processes.

1.4 Overview

Over the past several years, a valuable line of research has considered the impact of weakened number generators on cryptographic devices [19–23]. This research comprises three categories of work: (1) discovery of novel cryptographic attacks [8,19], (2) measurement and impact studies of known (theoretical) algorithm flaws [21] and (3) development of countermeasures and new theoretical models [20,22,23].

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 standards bodies to design or motivate analysis of new algorithms.

This work contains elements of all three categories, as identified in the below overview of each of the chapters herein.

- Chapter 2 provides background and context on PRG design, and a collated history of U.S. Federal cryptography standardization efforts as necessary to develop the study.
- Chapter 3 provides a starting point to observe how PRG standardization flaws can lead to critical failures in cryptosystems, and provides a first example of the disconnect between cryptographic best practices and knowledge, and final standards.

  It contains the first known instance of vulnerability detection using analysis of standard certification documentation. It is based on joint work [24] with Matthew Green and Nadia Heninger.

  The chapter is an example of category (2). I evaluate the *impact* of a specific flaw on real, deployed cryptographic systems. The results in this work demonstrate that these flaws are present and exploitable at scale in widely used implementations of real Internet protocols. The findings do not flow inevitably from previous results on ANSI X9.31 [8]. Moreover, this analysis is critical given that X9.31 is one of the most widely-deployed standard PRG algorithms in existence.

[19] Niels Ferguson and Dan Shumow. *CRYPTO*, 2007.
[20] Mihir Bellare, Kenneth G Paterson, and Phillip Rogaway. *CRYPTO*, 2014.
[21] Stephen Checkoway et al. *USENIX Security*, 2014.
[22] Yevgeniy Dodis et al. *EUROCRYPT*, 2015.
[23] Jean Paul Degabriele et al. *CRYPTO*, 2016.
[24] Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. *ACM CCS*, 2018.

- Chapter 4 shifts to look at adversarial design and usage of PRGs, through the lens of the Juniper Dual EC incident. The novel contribution in this chapter is an archaeology of the first known standards-based subversion of a PRG implementation. It strengthens the attack vector illustrated in the prior chapter by illustrating how an adversary can introduce flaws into a PRG in such a way that the backdoor is available only to adversarial party.

  It is based on joint work [25] with Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham.

  The chapter is again an example of category (2), evaluating the impact of a known cryptographic flaw in the context of a real world implementation.

- Chapter 5 indicates how restricting the threat vectors in scope of a standard can also lead to exploitable vulnerabilities by showing how the omission of side-channel threats from the FIPS standard permits the practical compromise of CTR_DRBG.

  It is based on joint work [26] with Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom.

  The chapter is an example of categories (1) and (2), developing novel cryptographic attacks as well as a real-world impact study following the methodology listed in Section 1.3 (A).

- Chapter 6 provides a comparative analysis of the different attacks developed in this work and provides practical recommendations for PRG design and use.

- Chapter 7 concludes this work by building a coherent picture of PSYCHOs and analyzing what characterizes the attackers and types of attacks that are feasible using the techniques illustrated in this work.

  This final chapter is an example of category (3), relying on the evidence provided in the prior chapters to establish its viability. The model is tested by evaluating other known attacks within the context of the model to determine if it is able to concisely describe a new class of threats.

[25] Stephen Checkoway et al. *ACM CCS*, 2016.
[26] Shaanan Cohney et al. *IEEE S&P*, 2020.

CHAPTER 2

BACKGROUND

This chapter provides background material on Pseudorandom Number Generators, and associated threat models.

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> *John von Neumann*

Real world deployments of cryptographic systems are generally reliant on a source of high quality randomness. To produce random data, such systems often implement a pseudorandom number generator (PRG): an algorithm that produces data indistinguishable from random. Failure of random number generators to perform this task correctly leads to the catastrophic failure of cryptosystems built on top of them. In this chapter I introduce a formal framework for evaluating the design of PRGs and their security.

2.1 Formalism

I begin by providing basic background regarding pseudorandom generators and their security properties. Informally, a PRG is an algorithm that given an initial seed, produces a stream of random bits such that an attacker cannot distinguish the produced stream from a truly uniform random bit stream with probability better than some negligible bound.

In this work I follow formalism based on Dodis et al.[9] and Woodage and Shumow[18], the latter of which was expressly designed to evaluate the security of recent NIST standardized PRGs.



A number of terms are used to denote pseudorandom number generators. These include Deterministic Random Bit Generator (DRBG), Random Number Generator (RNG), Pseudorandom Number Generator (PRNG) and PRG (with the same expansion). While a number of U.S. government standards use the term "DRBG", it does not seem to be widely used outside of the government context, so for the purposes of this work, I use the term pseudorandom generator (PRG).

Definition 1 (PRG wtih Input)**.** *A* PRG with Input *is a triplet of polynomial time deterministic algorithms* $\{\mathsf{instantiate}, \mathsf{generate}, \mathsf{reseed}\}$*. The PRG is instantiated by calling* instantiate *on an entropy sample* I *and a nonce* N*, and outputs initial state* S_0*. Next,* generate *gets as input a state* S*, a number of bits to output* $nbits$*, an additional input* $addin$*, and outputs new state* S' *and bits* $R \in \{0,1\}^{nbits}$*. Finally,* reseed *gets as input a state* S*, an entropy sample* I*, an additional input* $addin$*, and outputs a new state* S'*.*

Random Number Generation. *The PRG is prepared for use by a single call to* instantiate*. A user can then repeatedly request up to* r *random bits through a call to* generate*, which also outputs a new state for the PRG. Finally, both the user and the* generate *function can also call* update*, which updates the state of the PRG to a new state.*

PRG Security. Woodage and Shumow[18] define three security properties for a PRG: robustness, backtracking resistance, and prediction resistance. Backtracking resistance is the property that if the

generator is compromised at time t_1, an adversary remains unable to distinguish outputs generated prior to t_1 from truly random outputs. Similarly, prediction resistance ensures that there is some time t_2 after t_1 when no further outputs can be distinguished from truly random outputs. Robustness incorporates both of these guarantees into a single property.

While the models of Dodis et al. [9] and Woodage and Shumow [18] include an attacker that is able to compromise the entropy distribution used for sampling entropy to the PRG, this work considers a weaker attacker who is unable to do so. I therefore obtain stronger results as the weaker attacker is able break the PRG despite her inability to corrupt the entropy source.

This work instead assumes that the PRG correctly receives entropy samples drawn uniformly at random from the entropy space, better matching the real-world scenario.

Prediction Resistance. As mentioned above, prediction resistance models a PRG's ability to recover from state compromise. I begin by modeling an adversary capable of compromising the PRG state by allowing the adversary to execute the following procedures on the PRG:

- **get-output.** Models an attacker's ability to query the PRG for output.
  Calls $\mathsf{generate}(S, nbits, addin)$ where S is the current state, $nbits$ is the number of bits to output, and $addin$ is known by the attacker, and returns the output R.
- **set-state.** Models an attacker who compromises the state of the PRG. Gets as input an attacker-chosen value S^* and sets the PRG state $S \leftarrow S^*$.
- **next-ror.** Tests an attacker's ability to distinguish output from the PRG from uniformly random output. Sets $R_0 \leftarrow \mathsf{generate}(S, nbits, addin)$ with S as the PRG state, $nbits$ the number of bits in R_0, and $addin$ known by the attacker. It then sets R_1 to a value drawn uniformly at random from the same domain as R_0 and picks a choice bit $b \leftarrow^{\$} \{0, 1\}$ uniformly at random. The procedure returns R_b to the adversary which outputs a bit b'.

An adversary's advantage (and thereby the security strength of the PRG) is parameterized by the number of calls an adversary makes to the above procedures along with the adversary's probability of successfully guessing the challenge bit in the **next-ror** game. I use the following formal security definition for a PRG:

Definition 2 (Prediction Resistance)**.** *A PRG with input $\mathcal{G}$ is a $(t, q_D, q_R), \delta-$prediction-resistant PRG if for any adversary $\mathcal{A}$ running in time at most t, making at most q_D calls to* update *with q_R calls to next-ror/get-output, and one call to get-state, which is the last call $\mathcal{A}$ is allowed to make prior to calling next-ror, it holds that*

$$\left|\Pr\left[b = b' \mid b' \leftarrow \mathcal{A}_{\mathcal{G}}^{\mathsf{OP}}(q_D, q_R)\right] - 1/2\right| \leq \delta$$

where $\mathsf{OP} = \{$*next-ror*, *set-state*, *get-state*, *get-output*$\}$.

2.2 FIPS and FISMA

The Federal Information Security Management Act of 2002 ("FISMA") is a title of the E-Government Act of 2002 enacted by the US Congress to mandate federalized standards for information security. It places charges the National Institute of Standards and Technology (NIST) to develop "standards, guidelines, methods and techniques" [27] to "provide effective government wide management and oversight of the related information security risks, including coordination of information security efforts throughout the civilian, national security, and law enforcement communities" [28].

In order to establish such standards, NIST produces a series of publications under the moniker "Federal Information Processing Standards" (FIPS). The key cryptographic standard I investigate is FIPS 140-2, the successor to FIPS 140-1. The earlier standard was released July 17, 1995, establishing the program, and was superseded on May 25, 2001 by FIPS 140-2.FIPS 140-2 [29] defines requirements for cryptographic devices and software. This standard is applied within the Cryptographic Module Validation Program (CMVP), a program that certifies products used in US government applications.

[27] *15 U.S. Code §278g–3 - Computer standards program*
[28] *44 U.S. Code §3541*
[29] National Institute of Standards and Technology. *FIPS PUB 140-2 Security Requirements For Cryptographic Modules.* 2001

Compliant devices are eligible for certification under the program, which is jointly administered by NIST and the Communications Security Establishment (CSE) of Canada.

A prerequisite for the CMVP is the Cryptographic Algorithm Validation Program (CAVP), which tests individual algorithms and components as distinct from the CMVP which is analogous to a cryptographic 'integration test', testing the security of the overall composed module.

FIPS 140-2 defines four different levels of security, "Level 1" to "Level 4". Level 1, the lowest level, specifies no anti-tampering or other physical security measures. Level 4 requires that a cryptographic module should be able to detect tampering and zeroize keys if tampering is detected. At Level 2 and above, environment in which the module functions is required to be secured as to meet the specifications, and must be hardened against attack.

Vendor Certification The CMVP allows vendors to certify that their cryptographic modules meet minimal requirements to sell to the United States and Canadian governments. In order to comply with FIPS 140-2, implementations with a PRG must use an approved design, which are currently described in SP 800-90A, a publication described below in Section 2.2.2.

To certify a system, a firm with submits the final product to a testing lab (itself certified under a separate program administered by NIST). The submission requires vendors to provide product security specifications which detail the boundaries of the cryptographic module (within which processing occurs), the algorithms used, and how cryptographically sensitive material is controlled.

Once the vendor remediates any issues found to the satisfaction of the lab, NIST evaluates documentation provided by the lab and vendor before finalizing a certificate attesting that the product has met the requirements of the CMVP.

After a device completes the CMVP certification process it is added to a public list of approved devices that US federal agencies and other regulated bodies may use. The CMVP certificates and security policy are also posted publicly on NIST's webpage.

[30] Tammy Green et al. *ICMC*, 2019.

Certification is a time-consuming process, taking from a minimum of three months up to a maximum of two years from the time a vendor begins the process, to the date the certification is finalized [30]. There are also significant costs involved in documenting, testing, and submitting products for certification under the program.

Revalidation was until recently an arduous process that was required for even relatively minor product changes, dissuading vendors from patching issues. As discussed in Chapter 5, this was a factor in OpenSSL's decision not to remediate the flaw in their FIPS certified module following disclosure of the attack I identified.

Certification can apply narrowly to a specific product model, or apply to a product line. Most major vendors of network devices and operating systems certify their products.

I now review a series of PRGs that were standardized under FIPS.

2.2.1 X9.31

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 [31]. I analyze this PRG in Chapter 3.

2.2.2 NIST SP 800-90A

NIST Special Publication (SP) 800-90 is entitled "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" and is the de facto standard for algorithms for generating random numbers. The document was first published in 2006 and has undergone three revisions: "800-90 Revised", published in 2007, "800-90 A", published in 2012, and "800-90A Rev. 1", published in 2015. The first three publications contained four pseudorandom number generator designs, while the

[31] Elaine Barker and Allen Roginsky. *NIST SP 800-131A*, 2011.

PRG	Certificates
X9.31	946
CTR_DRBG	254
Hash_DRBG	208
HMAC_DRBG	124
Dual EC	81

Table 1: Number of CMVP Certificates Issued to PRGs as of Jan 1, 2014. This data was scraped from the official CMVP webpage. As shown, the X9.31 PRG was the most broadly adopted design as of 2014, followed by CTR_DRBG. X9.31 had a historical advantage, approved for a long period of time and, CTR_DRBG similarly centered around a block cipher (normally AES), could easily be substituted. Further, the implementation of AES in hardware could serve as a performance optimization, incentivizing use of the design. Dual EC notably suffered from limited adoption over its lifetime.

last publication contained only three. The missing design was the infamous Dual EC DRBG, which was removed from the publication in June 2015, years after Shumow and Ferguson discovered a design flaw that portended a backdoor[19], later given further credence by the Snowden disclosures [17]. The three remaining designs in NIST 800-90A Rev. 1 are HMAC_DRBG, HASH_DRBG and CTR_DRBG, which are based on HMAC, hash, and block cipher primitives respectively. The remainder of this work will refer to the 2015 publication as SP 800-90A unless otherwise noted.

2.2.3 Current and Historical Use

The only currently approved PRGs are HMAC_DRBG, Hash_DRBG, and CTR_DRBG as standardized in NIST SP 800-90A. The various approved designs have been adopted at significantly different rates.

In Table 1 I tabulate the number of cryptographic modules certified under the CMVP containing each PRG design as of the start of 2014, after which the X9.31 PRG and Dual EC were no longer approved for use. This provides a sense of the relative popularity of each of the PRGs. I provide analysis of PRGs in use today in Chapter 5.

2.3 Prior Work

The study of PRG design failures described in this work is preceded by the following works. I defer discussion of intentional design flaws to Chapter 4.

Cryptanalysis of PRGs There is a long history of cryptanalysis of practical pseudorandom number generator designs in the literature. A line of work beginning with Kelsey et al. [8] and continuing to today [9,12] has sought to formalize the security desiderata for PRGs used as part of cryptographic systems, and to evaluate deployed PRGs against these desiderata. Kelsey et al. [8] enumerate classes of attacks on PRGs, and note several design flaws and vulnerabilities against PRG designs, including the key compromise vulnerability in X9.17/X9.31 PRG that is explored in Chapter 3. Gutterman, Pinkas, and Reinman [32] and, later, Lacharme et al. [33] analyzed the Linux randomness system; Dorrendorf, Gutterman, and Pinkas [34] analyzed that of Windows.

Dodis et al. [9] defined a notion of recovery from state compromise for a PRG and from it showed that the Linux random number generator did not satisfy this definition. They showed by extension that there were inputs that would mislead the entropy estimation function and cause the PRG to fail to recover from state compromise. Michaelis, Meyer, and Schwenk [35] analyzed Java random number generation implementations and noted several vulnerabilities, including a vulnerability in Android.

Green [36] noted the dangers of using X9.31. He additionally highlighted the danger and usage of a global X9.31 key for the PRG in an early draft of AACS, the digital rights management specification for HD-DVD and Blu-ray distributions.

Kleptography Young and Yung [37,38] define *kleptography* which is a theoretical model of black box cryptographic backdoors that subverts systems by subliminally leaking information. Bellare, Paterson, and Rogaway [20] devise methods to secure symmetric cryptography against algorithm substitution attacks. Juels and Guajardo [39] develop immunization techniques using injections of randomness for the case of RSA key generation. Goh et al. [40] develop kleptographic techniques for leaking TLS session keys. Dodis

[32] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. *IEEE S&P*, 2006.
[33] Patrick Lacharme et al. *The Linux Pseudorandom Number Generator Revisited*. Cryptology ePrint Archive, Report 2012/251. 2012
[34] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. *TISSEC*, 2009.
[35] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. *CT-RSA*, 2013.
[36] Matthew Green. *Random number generation: An illustrated primer*. Online: https://blog.cryptographyengineering.com/2012/02/21/random-number-generation-illustrated/. Aug. 2016
[37] Adam Young and Moti Yung. *EUROCRYPT*, 1997.
[38] Adam Young and Moti Yung. *CRYPTO*, 1997.
[39] Ari Juels and Jorge Guajardo. *PKC*, 2002.
[40] Eu-Jin Goh et al. *ISC*, 2003.

et al.[22] formally model backdoored random number generators, design backdoored PRGs with strong indistinguishability properties, and evaluate countermeasures against backdoors. Degabriele et al.[23] build on this by giving efficient constructions such PRGs and bounding the duration of the compromise in terms of the state-size of the PRG. Stevens[41] developed techniques for "counter-cryptanalysis" that he used to reconstruct the MD5 collision attack that the unknown authors of the Flame malware exploited against the Microsoft Terminal Server Licensing Service prior to its discovery in 2012. This incident is the best prior example known of a publicly visible cryptanalytic attack carried out by sophisticated attackers.

Randomness Failures One category of PRG failures appears to be due to failure to properly seed a random number generator before use, or seeding with poor-quality inputs. Between 2006 and and 2008, Linux systems running the Debian distribution or its derivatives (including Ubuntu) shipped a modified version of the OpenSSL library that failed to incorporate entropy from the kernel into its own entropy pool. The available entropy was then low enough under normal conditions that the keys that affected systems generate could be exhaustively enumerated and identified over the network[42].

PRG failures have contributed to numerous real world vulnerabilities. In 1996, Goldberg and Wagner[43] showed that the Netscape browser seeded its PRG insecurely, allowing SSL traffic to be decrypted. In 2010, a group of hackers at the Chaos Communication Congress demonstrated an attack on the ECDSA private key used by the Sony PS3, that resulted from Sony's failure to properly generate new nonces[44].

In 2012, Heninger et al.[45] discovered a boot-time failure of the Linux random number generator to properly incorporate entropy sources on embedded and headless systems; this flaw allowed thee researchers to compute RSA private keys for 0.5% of TLS hosts and DSA private keys for 1.06% of SSH hosts in 2012 . Lenstra et al.[46] 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.

[41] Marc Stevens. *CRYPTO*, 2013.
[42] Scott Yilek et al. *IMC*, 2009.
[43] Ian Goldberg and David Wagner. *Dr. Dobb's Journal*, 1996.
[44] bushing et al. *CCC*, 2010.
[45] Nadia Heninger et al. *USENIX Security*, 2012.
[46] Arjen Lenstra et al. *CRYPTO*, 2012.

The root cause of both these results was the lack of entropy available shortly after boot in many network devices.

In 2016, Hastings, Fried, and Heninger [47] performed a follow-up study that found low to nonexistent software patching rates for systems affected by the 2012 PRG flaws. In a further study designed to measure the long term impact of their prior disclosures, Hastings, Fried, and Heninger [47], found very low patching rates and little improvement in systems affected by the Debian flaw. Kim, Han, and Lee [48] showed that a related problem affected OpenSSL on Android.

Nohl et al. [49] examine the feasibility of reverse engineering an RFID tag, and discovered PRG vulnerabilities that they exploited to obtain the embedded cryptographic keying material.

Bernstein et al. [50] were able to factor 184 keys from a sample of approximately 2 million smart card-generated RSA keys from generated for Taiwanese "Citizen Digital Certificate" smart card ID system. They hypothesized that the failures were due to a flawed hardware random number generator on some smart cards combined with a failure to whiten raw hardware PRG outputs. Kadianakis et al. [51] 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 PRG implementations. Ristenpart and Yilek [52] and Everspaugh et al. [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 [54].

[47] Marcella Hastings, Joshua Fried, and Nadia Heninger. *IMC*, 2016.
[48] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. *ACM CCS*, 2013.
[49] Karsten Nohl et al. *USENIX Security*, 2008.
[50] Daniel J. Bernstein et al. *ASIACRYPT*, 2013.
[51] George Kadianakis et al. *"Major key alert!" Anomalous keys in Tor relays*. Tech. rep.
[52] Thomas Ristenpart and Scott Yilek. *NDSS*, 2010.
[53] Adam Everspaugh et al. *IEEE S&P*, 2014.
[54] Alex Klyubin. *Some SecureRandom Thoughts*. Online: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html. Aug. 2013

CHAPTER 3

EARLY SIGNS OF TROUBLE

This chapter presents findings on the exploitability of the ANSI X9.31 pseudorandom number generator, an 1980s design that was included in certified products as late as 2014. I extend a growing line of research into weakened or backdoored random number generation in deployed products [19–23] which demonstrates the interplay between federal compliance efforts, and the security of the final system.

> This is our world now... the world of the electron and the switch, the beauty of the baud.
>
> The Mentor



Contributions

This chapter draws on work published in ACM CCS 2018 under the title *"Practical state recovery attacks against legacy RNG implementations"*, coauthored with Nadia Heninger and Matthew D. Green. I performed the entirety of the technical work for this publication, with advice, support and assistance in writing the paper provided by my co-authors. The idea to examine Fortigate products originated in collaboration with Matthew D. Green.

The chapter makes the following novel contributions:

- This work is the first to identify exploitable flaws in cryptographic devices by analyzing the output of cryptographic module validation procedures, demonstrating that existing standards validation procedures need to be revisited.
- I demonstrate the existence of widespread and *passively* exploitable vulnerabilities in implementations of one of the most widely-deployed standard pseudorandom number generators (PRGs) . The work includes a systematic study of FIPS 140 security policy documentation and identifies several independently vulnerable PRG implementations from different vendors, and discover critical failures in the standardization process.
- I develop an efficient passive X9.31 state recovery attack for the FortiOS v4 Internet Protocol Security (IPsec) implementation and demonstrate full IPsec VPN decryption. I use Internet-wide measurements to measure the scope of this vulnerability, and demonstrate the attack against against hosts in the wild, uncovering more than 25,000 vulnerable hosts.

3.1 Introduction

The first PRG to be standardized by the federal government was the ANSI X9.31 generator. This generator appeared in various publications beginning in the mid 1980s and shown earlier in Table 1 was the most commonly certified PRG until the mid 2010s. It first appeared as the ANSI X9.17 "Financial Institution Key Management (Wholesale)" standard, published in 1985 [55], which defined a voluntary interoperability standard for cryptographic key generation and distribution for the financial industry. The PRG appeared 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 US government cryptographic standards for the next three 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 published extensions of this design using 3-key 3DES and AES as the block cipher [56] that were 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 [8]. Perhaps due to this known weakness, the ANSI

[55] Accredited Standards Committee (ASC) X9, Financial Services. *ANSI X9.17: Financial Institution Key Management (Wholesale)*. Tech. rep. 1985

[56] Sharon 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

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. [8] in 1998, the NIST documents specifying the ANSI X9.31 PRG design fail to specify how the cipher key should be generated[56].

This raises the possibility that even FIPS-validated deployed systems could contain vulnerabilities that admit practical PRG state recovery. To evaluate this possibility I performed a systematic study of publicly available FIPS 140-2 certification for hundreds of products that implemented the ANSI X9.31 random number generator.

The work then explores the real world implications of this flaw, through the demonstrated exploitation of the FortiOS v4 network operating system. This operating system was distributed by Fortigate for their network devices, and was officially supported between 2009 to 2014.

My 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, I reverse-engineered the binaries for a Fortigate VPN gateway using FortiOS version 4. I 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 National Institute of Standards and Technology (NIST) Random Number Generator Validation System (RNGVS) specification[57], published as a validation suite alongside their standardization of the generator. I perform full state recovery in under a second from random number generator output. I 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. I demonstrate a full attack that learns the session keys for a Fortigate IPsec VPN using FortiOS version 4 in seconds. Furthermore, I demonstrate that this vulnerability exists *in the wild* by performing state

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

recovery, key recovery, and decryption on handshakes I 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 I 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.

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 [58] or hash functions [5]. In addition, as discussed in Section 6.1, the vulnerability could have been rendered practically unexploitable by using sufficiently high resolution timestamps.

A key differentiator between this work and previous work is that it 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 [25]. Moreover, my findings are tied to the analysis approach: they would not have been detectable through black-box external testing methods, such as those conducted by NIST as part of the Cryptographic Module Validation Program (CMVP).[i] To my knowledge, this work is the first to identify exploitable flaws in cryptographic devices by analyzing the output of cryptographic module validation procedures. This demonstrates that the existing standards validation procedures may need to be revisited.

3.2 Background

3.2.1 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 Federal Information Processing Standard (FIPS) and

[i] Unlike many previous PRG weaknesses, *e.g.,* [42,45] 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.

[58] Manuel Blum and Silvio Micali. *SIAM Journal on Computing*, 1984.

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 2005 [56]. While this design has appeared under various names, the work will refer to it as the X9.31 PRG for the remainder of this work, to use the terminology in modern implementations and standards.

The PRG is based on a block cipher with block size ℓ bits. I 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 [56] 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

$$R_i = E_K(I_i \oplus V_{i-1}) \tag{3.1}$$

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.

3.2.2 State Recovery Attack with a Known Key

I am 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.

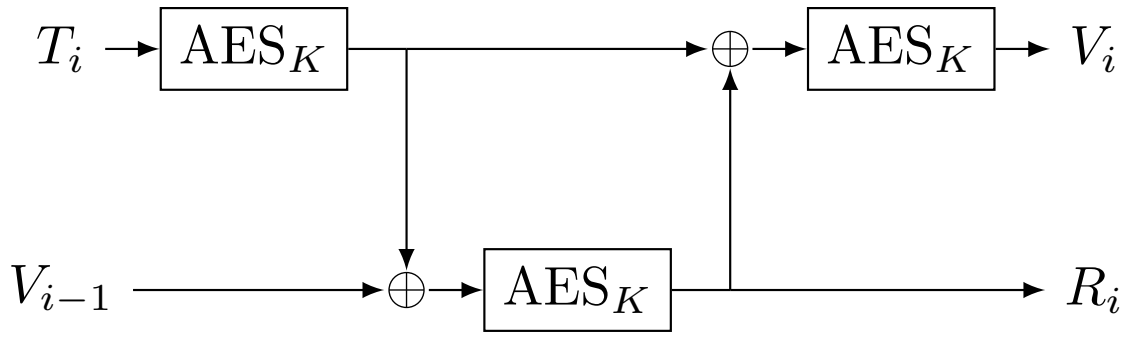


Figure 1: ANSI X9.31 PRG. Each iteration of the ANSI X9.31 PRG generation function (G) inputs a timestamp T_i and a seed V_{i-1} and produces an output block R_i and a new seed V_i.

Kelsey et al. [8] 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. One can relate these quantities as:

$$D(D(R_1) \oplus E(T_1)) = R_0 \oplus E(T_0) \tag{3.2}$$

If the timestamps are only known approximately, one can brute force them within some range until one finds a pair that yields equality, or apply a meet-in-the-middle attack [8]. If one block is not known completely, one 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:

$$R_2 = E(E(T_2) \oplus V_2) \tag{3.3}$$

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, I will describe how this attack works in theory in the context of popular cryptographic protocols.

3.2.3 Attacking X9.31 in TLS.

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

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.

3.2.4 Attacking X9.31 in IPsec

I describe the impact of a compromised random number generator on the IKE key exchange used in IPsec in the context of the attack on X9.31 PRG implementations.

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 [59] 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.

This work focus its 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 [59].

Each packet includes an 8-byte connection identifier called a cookie. The ISAKMP specification (RFC 2408) [60] suggests generating the cookie 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 point

[59] D Harkins and D Carrel. *The Internet Key Exchange (IKE)*. RFC 2409. Internet Engineering Task Force, 1998

[60] M Schneider D. Maughan M. Schertler and J Turner. *Internet Security Association and Key Management Protocol*. RFC 2408. Internet Engineering Task Force, 1998

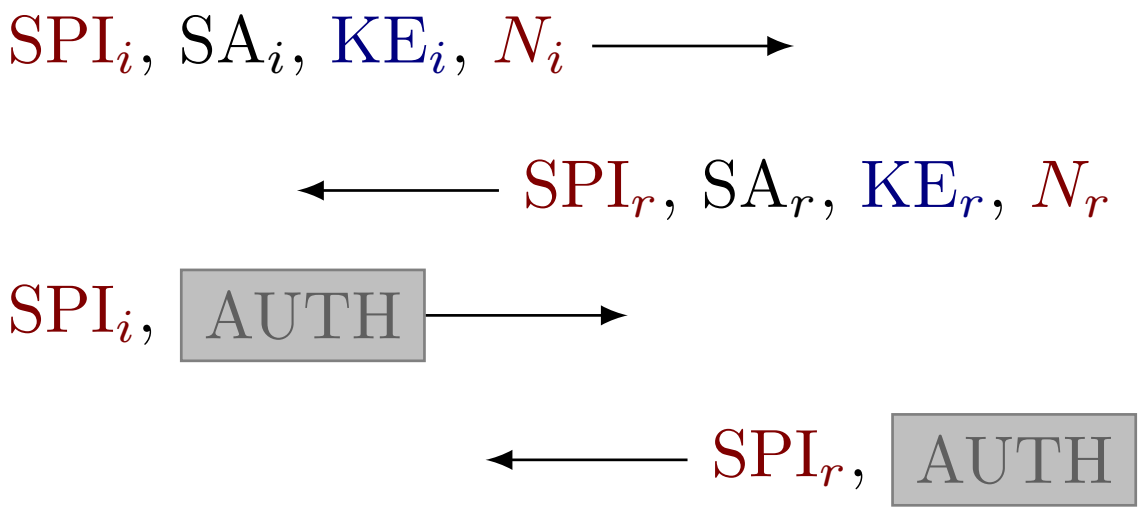


Figure 2: Randomness and the IKEv2 Handshake. The IKEv2 handshake establishes an authenticated, encrypted connection using a Diffie–Hellman key exchange. In the target implementation, both the SPI and nonce N are raw, unencrypted outputs from the PRG. The key exchange message KE is generated from the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

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 [61]. I show an abbreviated version of the IKEv2 handshake in Figure 2. First the initiator sends an IKE_SA_INIT message, with proposals similar to

[61] C. Kaufman. *Internet Key Exchange (IKEv2) Protocol.* RFC 4306. Internet Engineering Task Force, 2005

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. [ii]

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.

State recovery in IPsec. 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 (3.2). The attacker then guesses the two timestamps used for the next two blocks of output using Equation (3.3), 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. I

[ii] 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.

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.

3.3 FIPS and Hard-coded X9.31 Keys

As discussed in Section 3.2.1, the NIST design description for the X9.31 random number generator [56] 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. I 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. I obtained the list of certified devices from the NIST web site [62].

3.3.1 Certified unsafe usage of the X9.31 PRG

I 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 [63]. 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.

[62] National Institute of Standards and Technology. *CMVP Historical Validation List*. Feb. 2017

[63] Computer Security Resource Center, National Institute of Standards and Technology. *Cryptographic Algorithm Validation Program - RNG Validation List*. Online: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/validation/validation-list/rng

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

Table 2: Certificate issuances for X9.31. Issuance continued even after the publication of SP 800-90 in 2006. The first SP 800-90 CMVP certifications were issued in 2008, yet 47% of FIPS certificates issued 2008-2016 were for X9.31.

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 3: 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.

The security policy documents each contain a list of Critical Security Parameters (CSPs), which includes access control, key and parameter generation, and zeroization policies. I 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 I was unable to determine whether the key was generated securely, I exclude these from further study. This left 161 vendors who mention seed key generation in some capacity.

I 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. I did not study these cases further. The largest class of devices I 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 I considered to be safe X9.31 key generation, the InZero Gateway security policy states that the "PRG is seeded from `/dev/urandom`..." this provides the PRG with 256 bits of entropy for the seed key"[64]. 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[45], I 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[65], which was certified for the X9.31 generator, states that for the "PRG 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.

I 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. I list these products together with the language used to describe seed key generation in Table 3.

3.3.2 Device-specific analysis

I was only able to gain access to the binary image for one of the products I identified as potentially vulnerable, a Fortinet operating system. I give more details on my 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 hard-coded key, although they agreed with my interpretation of the certification language. They informed us that the 4400 controllers reached end of support in 2016, the

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

[65] JUNIPER NETWORKS, INC. *FIPS 140-2 SECURITY POLICY - SSG 140.* Online: https://www.juniper.net/documentation/hardware/netscreen-certifications/Security_Policy_SSG-140_ScreenOS_6_2.pdf

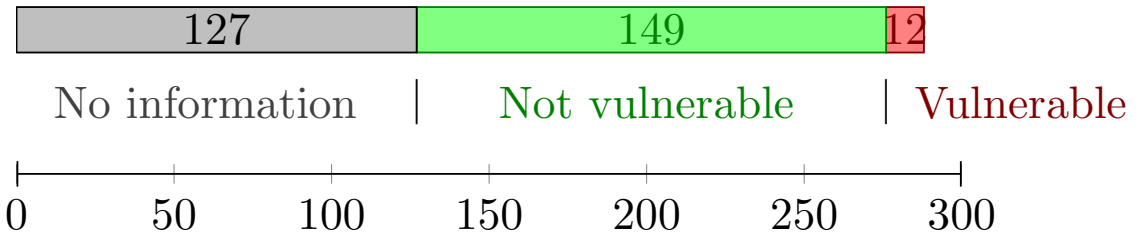


Figure 3: Counting vulnerable implementations. I examined the security policy documents from 288 vendors who had been FIPS 140-2 certified for the X9.31 PRG for information on how the seed key for the random number generator was generated. 12 vendors, or 4% of the total, documented a hard-coded key vulnerability.

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, 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.

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

Table 4: Affected Implementation Versions.

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

3.3.3 Open source implementations

I 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.

3.4 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". I 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.

I 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 my hardware, after which the attacker is able to guess the secret keys used to generate encryption keys. I performed an Internet-wide scan for affected hosts, and were able to carry out state recovery and private key recovery on handshakes from my scan data.

3.4.1 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[66]. 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 my 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 my disclosure of the random number generation vulnerability[67], Fortigate released version 4.3.19 of FortiOS in November 2016.

Vulnerabilities in FortiOS. On January 15, 2016, the MITRE corporation posted CVE-2016-1909 [68] revealing the presence of a hard-coded 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 [69], 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". Analyses by private firms, including the Russian security firm Kaspersky indicate links between The Equation Group and the National Security Agency (NSA) [70]. Among other things, the leak contained a remote code

[66] IDC 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: https://www.idc.com/getdoc.jsp?containerId=prUS41490016

[67] MITRE Corporation. *CVE-2016-8492*

[68] MITRE Corporation. *CVE-2016-1909*

[69] Fortinet. *Brief Statement Regarding Issues Found with FortiOS*. Online: https://blog.fortinet.com/2016/01/12/brief-statement-regarding-issues-found-with-fortios. Jan. 2016

[70] Kaspersky Labs. *Equation Group Questions and Answers*. https://web.archive.org/web/20150217023145/https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf. 2015

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 [71]. I did not find any evidence in the Shadow Brokers leak that The Equation Group was aware of the vulnerability I found in the PRG.

3.4.2 Static Analysis

I 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 3.5.

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.

3.4.3 Debugging Instrumentation

To aid in the debugging process, I wrote a kernel module that logged state variables like the time, the internal PRG state, and the raw output to the kernel log.

For the virtual machine, I used the hypervisor's debugging interface to read program states from memory directly.

For the hardware device there was no available debugging interface. Instead, I found an exploit on the operating system that gave us a root shell. FortiOS v4 provides an interface on their custom shell

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

to execute a number of preset utilities `/bin` on the filesystem. These utilities were all symbolic links to a binary that executes different programs depending on the name of the link, which is provided as the zeroth argument by the operating system. This is a similar design to BusyBox [72]. The non-linked source was the init binary, which contained within it all the programs. This shell provides a number of undocumented commands prefixed by '`fnsysctl`' including `ls`, `ln` and `ftp`. The `ftp` command points to a non-existent binary.

The Fortigate 100D has two auto-mounted USB ports. I loaded a statically compiled shell onto a USB drive and used `ln` to link it to `/bin/ftp`. I could then use the built-in `ftp` command to execute it with root privileges. I developed the shell and other binaries used in the exploitation process in a virtual machine running Slackware 11, which uses a Linux kernel (2.4.33.3) close to the old kernels used by the Fortigate devices (2.4.37).

3.4.4 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.

I reverse engineered the kernel module providing the X9.31 implementation and found the hard-coded AES key used for the PRG. (See Listing 3.1 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 [57].

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.

```
1 int initialize_X931()
```

[72] Denys Vlasenko. *BusyBox: The Swiss Army Knife of Embedded Linux*. Online: https://busybox.net/about.html. 2008

```
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
16         clarity */
17     save_state(rng_state);
18     fill_timestamp(timestamp_buffer);
19     result =
20        x931(&timestamp_buffer, output_buffer,
21                     rng_state, 16);
22   }
23   return result;
24 }
```

Listing 3.1: The X9.31 Initialization Routine.

3.4.5 The HTTPS Implementation

I 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.

3.4.6 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" [73]. 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.

I generated a certificate on the VM and verified that the most significant bits of the RSA factors are related by Equation (3.2). 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.

3.4.7 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. I 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.

[73] National Institute of Standards and Technology. *FIPS PUB 186-4: Digital Signature Standard (DSS)*. 2013

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 [iii].

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 I was unable to reverse engineer, and outputs the result of the modular exponentiation.

I was able to invoke this system call ourselves on the hardware device to generate the Diffie–Hellman public key exchange values and shared secrets from candidate PRG blocks.

3.4.8 State recovery in IKEv1

The state recovery attack outlined in Section 3.2.2 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 a second, so I assume it is known. However, the heap-allocated pointer provides approximately 13 bits of entropy [74]. Rearranging Equation (3.2), 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

[iii] The IKEv1 cookie was $\texttt{SHA1}(\texttt{0x2020}||\texttt{mpz_d}||\texttt{src}||\texttt{dst}||\texttt{timestamp}||\texttt{nonce}_{16})$. I 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.

[74] William Herlands, Thomas Hobson, and Paula Donovan. *USENIX Workshop on Cyber Security Experimentation and Test*, 2014.

second block of PRG output R_1 is known, and I estimated a 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, I found that IKEv2 state recovery was cheaper, and focused my 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.

3.4.9 State recovery in IKEv2

As discussed in Section 3.2.4, in order to recover the PRG state, one needs 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. I learned the static key as described above by reverse-engineering the source code. I use the capture time of an incoming handshake to approximate the timestamps. From these, I use the approach described in Section 3.2.2 to recover the PRG state. I found that searching within a one-second window, or about 2^{21} guesses for the first timestamp, worked well on the hardware as well as scanned machines in the wild. [iv]

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

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 the 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 the hardware appliance in under one second on 12 cores of an Intel Xeon E5-2699 with parameters as above. For an expanded search space

[iv] 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.

of 100 microseconds for the second timestamp as described in Section 3.5, successful runs completed in an average of 15 core minutes. Although verification is performed against only half a block of raw output from the first block, the reduced size of my timestamp search means that the expected number of false positive matches in Equation (3.2) is small, with probability at most $2^{26}/2^{64} = 2^{-38}$.

3.4.10 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 the search. Although one verifies Equation (3.2) 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}, I expect a false positive with probability $2^{26}/2^{96} = 2^{-70}$.

3.4.11 Recovering the IKEv2 Keys

Once one has recovered the PRG state from the SPI (block R_0 of output) and nonce (block R_1 of output), one can then wind forward the PRG, successively guessing the two following timestamps and applying Equation (3.1) to recover two more blocks R_2 and R_3 that will be used to generate the Diffie–Hellman secret.

One calculates $g^{R_2||R_3} \bmod p$ (where $||$ denotes concatenation) and checks this value against the Diffie–Hellman public value in the IKE_SA_INIT packet until one finds a match.

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

I 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. 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, I set the 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 I was not able to reverse-engineer, my 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.

3.4.12 Recovering Traffic Keys

Once one recovers the victim device's public key value, one can make another call to the ASIC with the 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. I computed the SKEYSEED as described in Section 3.2.4 and verified full passive decryption against traffic to the FortiGate 100D.

3.5 Measurements

I 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.

3.5.1 HTTPS

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

[v] This is lower than the approximately 40 million HTTPS hosts seen in scans offering a wider variety of SSL/TLS versions and cipher suites.

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

(a) Supported TLS Cipher Suites 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			

(b) Supported IKEv2 Parameters in FortiOS v4.

Table 5: Scanning Parameters Used to Find FortiOS Devices.

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

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. I found 114,172 hosts with a matching certificate organization field; their common names identified 3,379 unique model numbers.

State recovery. The state recovery attack was successful against 23,517 hosts, or 20.6% of hosts with default Fortinet certificates. I attempted state recovery using a 1s window around the time encoded in the server random. Figure 4 shows the distribution of the number of timestamps guessed for successful state recoveries. In Figure 5a, I 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 5b shows the distribution of the gap between the timestamps for the first and second PRG blocks in the TLS server random. I brute forced up to an offset of 100 μs after the first timestamp, but all the observed state recoveries had a gap of no more than 40 μs between the first and second timestamps.

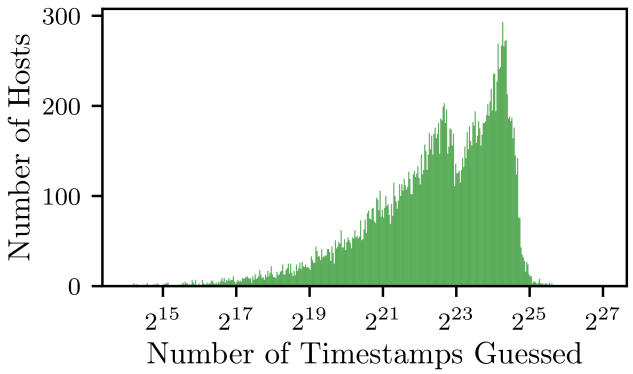


Figure 4: Brute force work for TLS state recovery. The average number of timestamp guesses required for each host was $2^{22.9}$ with a standard deviation of $2^{23.1}$.

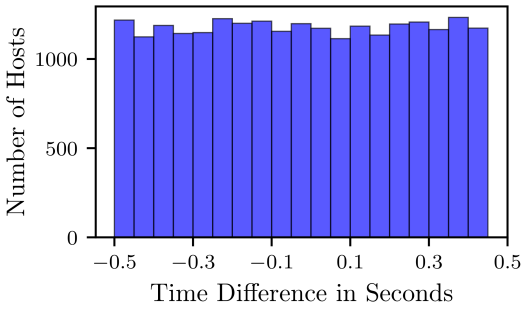
Specific HTTP files. My hardware device's administration panel contained an image file located at `/IMAGES/LOGON.GIF` that I used as a rudimentary fingerprint for FortiOS. In the HTTPS scan, I sent a GET request for this file. 605,950 hosts responded with `HTTP OK` and a corresponding image. The others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

ETag headers. The HTTP ETag header uniquely identifies HTTP server resources, and is used for web cache validation along with conditional requests[75]. 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, I matched headers from the scan against known ETags for FortiOS v4.

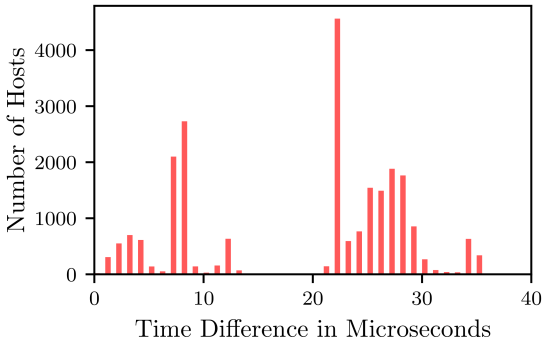
The Equation Group leak[76] contained a list of 440 ETag suffixes for some FortiOS device and firmware-build pairs, including 168 entries corresponding to 9 models and 26 builds of FortiOS v4. The leak also contained a memory address for each entry, used for the Egregious Blunder exploit with which it was packaged. The ETag for the FortiGate 100D (`5192DBFD`) was not included in the database, so I added it to the search.

[75] John Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. Internet Engineering Task Force, June 1999

[76] Dan Goodin. *Group claims to hack NSA-tied hackers, posts exploits as proof*. Aug. 2016



(a) 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 the successful state recovery trials (at µs resolution) was 510µs.



(b) Subsesquent timestamp offset. I 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: Recovering Timestamps from TLS Handshakes.

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

RSA public keys. Fortigate devices were already known to generate RSA moduli that share common factors [45,47]. My co-authors 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 by request to Hastings, Fried, and Heninger [47]. This yielded prime factors for 3,163 keys from my scan. However, the X9.31 state recovery attack described in Section 3.4.6, 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.

Limitations. The devices that responded to the HTTPS scans exposed the administration panel on a public IPv4 address, which is not the default configuration for FortiOS. My scans therefore identified only

those devices that were *misconfigured*. A correct (and default) configuration prevents my scanning but does not mitigate the vulnerability. The total population of vulnerable hosts is therefore likely *significantly* higher than the population visible to my 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.

3.5.2 IKEv2

I 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 Autonomous System 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, the key recovery attack succeeded against 7.

Cipher support. I sent handshake requests with cipher proposals that were supported in FortiOS v4, listed in Table 5b. I received 7,743,876 responses.

Nonce size. FortiOS v4's IKEv2 implementation uses 128-bit nonces. From the successful IKEv2 handshakes above, 50,285 had 128-bit nonces. I attempted state and key recovery from the handshakes with all of these hosts, and were able to successfully recover the Diffie–Hellman shared secret in the handshake I negotiated for 7 hosts using the key recovery attack I described above. This included 4 hosts that weren't seen in the population of vulnerable HTTPS hosts. I 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 my scans.

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

Table 6: X9.31 State and Key Recovery in the Wild.

Limitations. The number of IKE responses I received 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.

3.5.3 NOBUS and symmetric backdoors.

As I note in Section 3.1, the vulnerability I 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 as will be discussed in Chapter 4, where only the party who generated the 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 I 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.

CHAPTER 4

THE EVIL INCLINATION

This chapter discusses the Dual EC pseudorandom number generator, the backdoor discovered in the standard, and the subsequent modification of the backdoor parameters by an unknown adversary in Juniper's ScreenOS implementation. It develops a primordial framework for adversarial manipulation of standards, that is expanded on in the concluding chapter.

Th' infernal Serpent; he it was, whose guile
Stird up with Envy and Revenge, deceiv'd

Milton



Contributions

This chapter draws on work published in ACM CCS 2016 under the title *"A Systematic Analysis of the Juniper Dual EC Incident"*, coauthored with Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Matthew D. Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. My contributions to the work were in early unpacking of the Juniper update binary format, a portion of the reverse engineering, and work facilitating internet-wide scanning.

The chapter makes the following novel contributions:

- I demonstrate the passive exploitability of a vulnerability in another National Institute of Standards and Technology (NIST) standardized pseudorandom number generator (PRG), noting that following NIST guidelines is insufficient to protect implementers. While a cascade of software flaws was necessary for the underlying PRG vulnerability to be exploitable, NIST certification and testing procedures did not guard against them.
- Though past work demonstrated the exploitability of the PRG in the TLS cryptosystem, this is the first demonstration of an attack against a full implementation.
- I reconstruct a history of the set of changes necessary to the ScreenOS software platform that resulted in the vulnerability, and illustrate how powerful adversaries can manipulate both design, and implementation to cause critical breaks in secure systems. This does *not* implicate one specific party as having been responsible for all the flaws studied.

The Dual EC PRG was another of the designs standardized as part of modernization efforts; a design with a particularly fraught history.

In the early 2000s, the NSA pushed the ANSI standardization committee to include the PRG as part of a new standard, ANSI X9.82: Random Number Generation. This standard was intended to supplant the earlier ANSI recommended PRGs with more sophisticated designs, drawing from lessons learned

in the intervening years. In 2004, the third draft of X9.82 was published, including the PRG in Part 3. NIST soon followed suit, drawing heavily from the ANSI standard in its 2005 draft of SP 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators.

Prior to the completion of the standardization process, various authors identified serious concerns with the design. First, two concurrent works proved that a bit predictor could be made to distinguish the output bits from random [77,78]. The following year Ferguson and Shumow [19] presented a practical attack showing that an attacker with knowledge of 'trapdoor parameters' and two output blocks can recover the internal state of the PRG, and is subsequently able to predict all future output. In 2014, Checkoway et al. [21] demonstrated how the vulnerability could be used to effectively build an asymmetric backdoor into the SSL/TLS protocol.

I present analysis of Juniper Technology's network device operating system ScreenOS which contained a vulnerable instantiation of the Dual EC PRG. This chapter demonstrates a passive key recovery against the IPSec/IKE protocol suite, and discuss scanning methodologies for detecting vulnerable devices. The analysis includes a discussion of how a series of flaws that enabled the exploit came to be present, and determines that it was likely the work of adversarial parties.

The chapter is presented from the perspective of the full set of co-authors.

4.1 Introduction

In his statement for the record before the Senate Armed Services Committee on February 9, 2016, James Clapper, the U.S. Director of National Intelligence, illustrated the "worldwide threat assessment of the U.S. intelligence community" with an example of vulnerable infrastructure:

> A major U.S. network equipment manufacturer acknowledged last December that someone repeatedly gained access to its network to change source code in order to make its products'

[77] Berry Schoenmakers and Andrey Sidorenko. *Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator*. Cryptology ePrint Archive, Report 2006/190. 2006

[78] Kristian Gjøsteen. *Comments on Dual-EC-DRBG/NIST SP 800-90, Draft December 2005*. Mar. 2006

> default encryption breakable. The intruders also introduced a default password to enable undetected access to some target networks worldwide [79].

The "network equipment manufacturer" was Juniper Networks; it had disclosed the two issues in a security bulletin on December 17, 2015 [80], and released patched versions of ScreenOS, the operating system powering the affected NetScreen devices.

Immediately following Juniper's advisory, security researchers around the world — including our team — began examining the ScreenOS firmware to find the vulnerabilities Juniper claimed to have patched. They found that the change that, per Clapper, rendered ScreenOS encryption "breakable" did nothing but replace a few embedded constants.

We explain how these changed constants may have allowed whoever introduced them to decrypt passively recorded VPN traffic to affected devices. The 2012 change took advantage of Juniper's 2008 overhaul of the ScreenOS randomness which introduced the NSA-designed Dual EC random number generator, and included Juniper-selected constants which we are unable to verify are secure. Juniper's December 2015 patch restored these original constants.

Our methods include forensic reverse engineering of dozens of ScreenOS firmware revisions stretching back nearly a decade; experimental testing on NetScreen hardware; and Internet measurement studies.

Juniper's NetScreen devices were FIPS certified, and the affected code implemented the standard IPsec protocol suite. Our findings thus have implications for many of the stakeholders in the development of cryptographic products, including protocol designers, implementers, code reviewers, and policymakers.

Dual EC The NSA-designed Dual EC PRG [5,81] has the problematic property that an attacker who knows the discrete logarithm of one of the input parameters (Q) with respect to a generator point, and is able to observe a small number of consecutive bytes from the PRG, can then compute the internal

[79] James R. Clapper. *Worldwide Threat Assessment of the U.S. Intelligence Community*. Statement for the record, Senate Armed Services Committee. Online: http://www.armed-services.senate.gov/imo/media/doc/Clapper_02-09-16.pdf. Feb. 2016

[80] Juniper. *2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756)*. Online: https://kb.juniper.net/InfoCenter/index?page=content{&}id=JSA10713. Dec. 2015

[81] John Kelsey. *Dual EC in X9.82 and SP 800-90*. https://csrc.nist.gov/csrc/media/projects/crypto-standards-development-process/documents/dualec_in_x982_and_sp800-90.pdf. May 2014

state of the generator and thus predict all future output. The changed constants in ScreenOS defined this Dual EC point Q.

Summary of our findings Our analysis shows that from 2008 until it was patched in April 2016, Juniper's ScreenOS PRG implementation was vulnerable to efficient state recovery attacks conducted by an attacker who could select the Q value. This capability is not an inevitable result of the known attacks on Dual EC, but instead stems from a collection of design choices made by Juniper in 2008.

We identified a constellation of changes made to both the PRG and IKE implementations between ScreenOS 6.1 and 6.2 that together substantially predispose the IKE/IPsec implementation to state recovery attacks on the Dual EC generator. These changes, which were introduced concurrently with the addition of Dual EC, create a "perfect storm" of vulnerabilities that combine to enable a highly effective *single-handshake* exploit against the ScreenOS IKE implementation. Moreover, we identify several implementation decisions that superficially appear to reduce exploitability, but that on closer examination actually facilitate the attack. It is these changes — crucially, *not* the "unauthorized code" introduced by the third party — that enable passive VPN decryption for an attacker who has the discrete log of Q.

To validate the accuracy of our findings, we implement a proof of concept exploit against our SSG-550M running ScreenOS 6.3.0 and show that when the device is configured with a Q parameter of our choosing, our attacks can efficiently decrypt VPN connections from a single handshake, without seeing any other traffic. Moreover, we investigate the impact of different IPsec versions and configurations on the attack, and show that configuration decisions can substantially affect the exploitability of the device — in some cases completely compromising the device's security.

4.2 Dual EC

In this section, we describe the Dual EC pseudorandom number generator along with some details on how ScreenOS implements Dual EC. We also describe the attack on Dual EC due to Ferguson and Shumow [19].

Dual EC comes in a variety of forms. There are two slightly different NIST standards for Dual EC, which also contain optional features. There are three standard elliptic curves which can be used, and implementors are free to make a number of software engineering choices. Each of these design decisions can affect the difficulty of the Shumow–Ferguson attack. For concreteness, we describe Dual EC as implemented in Juniper's ScreenOS below. For more details on other forms of Dual EC, see Checkoway et al. [21].

Dual EC has three public parameters: the elliptic curve and two distinct points on the curve called P and Q. ScreenOS uses the elliptic curve P-256 and sets P to be P-256's standard generator as specified in NIST Special Publication 800-90A [82]. That standard also specifies the Q to use, but ScreenOS uses Juniper's own elliptic curve point instead. The finite field over which P-256 is defined has roughly 2^{256} elements. Points on P-256 consist of pairs of 256-bit numbers (x, y) that satisfy the elliptic curve equation. The internal state of Dual EC is a single 256-bit number s.

In ScreenOS, Dual EC is always used to generate 32 bytes of output at a time. Let $x(\cdot)$ be the function that returns the x-coordinate of an elliptic curve point; $\|$ be concatenation; $\text{lsb}_n(\cdot)$ be the function that returns the least-significant n bytes of its input in big-endian order; and $\text{msb}_n(\cdot)$ be the function that returns the most-significant n bytes. Starting with an initial state s_0, Dual EC generates 32 pseudorandom bytes *output* and a new state s_2 as follows,

$$\begin{aligned} s_1 &= x(s_0 P) & r_1 &= x(s_1 Q) \\ s_2 &= x(s_1 P) & r_2 &= x(s_2 Q) \\ \textit{output} &= \text{lsb}_{30}(r_1) \,\|\, \text{msb}_2\big(\text{lsb}_{30}(r_2)\big), \end{aligned}$$

where sP and sQ denote scalar multiplication.

In 2007, Ferguson and Shumow [19] noted that if the elliptic-curve discrete logarithm $e = \log_P Q$ (i.e., the integer e such that $eP = Q$) were known, then seeing *output* would reveal the Dual EC internal

[82] National Institute of Standards and Technology. *NIST Opens Draft Special Publication 800-90A, Recommendation For Random Number Generation Using Deterministic Random Bit Generators for Review and Comment*. Sept. 2013

state. The key insight is that one can obtain $d = \log_Q P = e^{-1} \bmod n$, where n is the group order, and then multiplying the point $s_1 Q$ by d yields the internal state $x(d \cdot s_1 Q) = x(s_1 P) = s_2$. Although $s_1 Q$ is itself not known, 30 of the 32 bytes of its x-coordinate (namely r_1) is the first 30-bytes of *output*.

This insight gives rise to the simple procedure to recover s_2. For each of the 2^{16} 256-bit integers r such that $\text{lsb}_{30}(r)$ equals the first 30-bytes of *output*, check if r is a valid x-coordinate of a point on the curve.[vi] In other words, find a point R such that $x(R) = r$. Roughly half of the r values will be valid x-coordinates.[vii] For each such R, compute $s' = x(dR)$ and $r' = x(s'Q)$. If the correct $r = r_1$ is chosen, $\text{msb}_2\big(\text{lsb}_{30}(r')\big)$ will be equal to the last two bytes of *output* and $s' = s_2$, the new internal state.

The one complication with the above procedure is that there may be several values of r such that $\text{msb}_2\big(\text{lsb}_{30}(r')\big) = \text{lsb}_2(\textit{output})$ and each such r corresponds to a potential internal state s'. In practice, this is a minor complication as it is exceedingly rare for there to be more than three such r.

4.3 History of the Juniper Incident

After NIST recommended against the use of Dual EC [82] in response to post-Snowden concerns about the default value of Q, Juniper published a knowledge base article [83] explaining their use of Dual EC in ScreenOS, the operating system powering its NetScreen firewall appliances, stating that although those products used Dual EC:

> ScreenOS does make use of the Dual_EC_DRBG standard, but is designed to not use Dual_EC_DBRG as its primary random number generator. ScreenOS uses it in a way that should not be vulnerable to the possible issue that has been brought to light. Instead of using the NIST recommended curve points it uses self-generated basis points and then takes the output as an input to FIPS/ANSI X.9.31 (sic) PRG, which is the random number generator used in ScreenOS cryptographic operations.

[vi] This procedure is sometimes called point decompression.

[vii] Each r that is an x-coordinate of some point R is also an x-coordinate of the point $-R$. It doesn't matter which point is chosen as R and $-R$ differ only in the "sign" of their y-component.

[83] Juniper Networks. *Juniper Networks Product Information about Dual_EC_DRBG*. Knowledge Base Article KB28205. Oct. 2013

The first of these mitigations — self-generated basis points[viii] — is not completely satisfactory because it depends on Juniper generating Q in such a way that nobody knows its discrete log, which they have not verifiably demonstrated. However, the second mitigation — if implemented correctly — defends against the current publicly known attacks on Dual EC because those attacks rely on having Dual EC output rather than a one-way function of that output, so even an attacker who knew the discrete log of Q would be unable to recover the PRG state.

This was the situation on December 17, 2015 when Juniper issued an out-of-cycle security bulletin [80] for two security issues in ScreenOS:

- CVE-2015-7755[ix] ("Administrative Access")
- CVE-2015-7756[x] ("VPN Decryption")

This announcement was particularly interesting because it was not the usual report of developer error, but rather of malicious code which had been inserted into ScreenOS by an unknown attacker:

> During a recent internal code review, Juniper discovered unauthorized code in ScreenOS that could allow a knowledgeable attacker to gain administrative access to NetScreen® devices and to decrypt VPN connections. Once we identified these vulnerabilities, we launched an investigation into the matter, and worked to develop and issue patched releases for the latest versions of ScreenOS.

The "Administrative Access" vulnerability was determined to be a back door in the SSH [84] daemon that would have allowed anyone who knew the correct password to log in with administrative access. This issue has been extensively discussed by Moore [85]. The second issue, however, turns out to be far more technically interesting. According to Juniper's advisory:

[viii] While the Juniper article says "points", actually only Q differs from the NIST default values. Juniper's implementation uses the default P value.

[ix] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7755

[x] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7756

[84] T Ylonen and C Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Internet Engineering Task Force, Jan. 2006

[85] H D Moore. *CVE-2015-7755: Juniper ScreenOS Authentication*. Dec. 2016

> VPN Decryption (CVE-2015-7756) may allow a knowledgeable attacker who can monitor VPN traffic to decrypt that traffic. It is independent of the first issue.
>
> This issue affects ScreenOS 6.2.0r15 through 6.2.0r18 and 6.3.0r12 through 6.3.0r20. No other Juniper products or versions of ScreenOS are affected by this issue.
>
> There is no way to detect that this vulnerability was exploited.

While both Juniper's advisory and the CVE itself are short on details, comparison of the binaries for the vulnerable and patched versions reveal that the relevant change to the code is a change in the value of Q and the corresponding test vectors [86]. The natural inference, therefore, is that the attacker changed Q away from Juniper's original version (which is itself different from the default Q in the standard) and that the patched version changes it back. What makes this even more interesting is that — as noted above — even a Q value for which the attacker knows the discrete log should not lead to a passive decryption vulnerability because the output is supposed to be filtered through the ANSI X9.31 PRG. This obviously raises serious questions about the accuracy of Juniper's 2013 description of their system, specifically:

1. Why does a change in Q result in a passive VPN decryption vulnerability?
2. Why doesn't Juniper's use of X9.31 protect their system against compromise of Q?
3. What is the history of the PRG code in ScreenOS?
4. How was Juniper's Q value generated?
5. Is the version of ScreenOS with Juniper's authorized Q vulnerable to attack?

Juniper's public statements regarding this incident set up a mystery. We have a body of material — an archive of firmware releases — that we use to answer many of the questions that make up that mystery. Specifically, we answer why changing Q matters and why X9.31 does not defend against that change in Sections 4.4 and 4.5. By examining multiple versions of ScreenOS, we reconstruct a rough timeline of the evolution of ScreenOS's PRG code in Section 4.9. Then there are some lingering questions, especially ones to do with motive or the identity of actors, that would require a different body of material to approach. We call these out in Section 4.9.2.

[86] Ralf-Philipp Weinmann. *Some Analysis of the Backdoored Backdoor*. Online: https://rpw.sh/blog/2015/12/21/the-backdoored-backdoor/. Dec. 2015

4.4 The ScreenOS PRG

In this section, we describe the methodology and results of our analysis of the ScreenOS 6.2 PRG cascade subroutines.[xi]

Immediately following Juniper's 2015 announcement [80], security researchers around the world — including our team — began examining the ScreenOS firmware in an attempt to find the vulnerabilities Juniper claimed to have patched. HD Moore quickly posted a diff of the strings in the firmware between versions 6.2.0r14.0 and 6.2.0r15.0 — the version where the unauthorized code was introduced.[xii] Line 934 shows a changed 32-byte hexadecimal constant. Based on the following hexadecimal constants, we hypothesized that this was the x-coordinate of an elliptic curve point on NIST curve P-256. By reverse engineering a copy of the firmware, it soon became apparent to our team and others that the changed constant was the x-coordinate of a nonstandard point Q in Dual EC.[xiii] Knowing that a third party changed the point Q in Dual EC is suggestive — but not dispositive — of that being key to being able to decrypt VPN connections described in Juniper's announcement. Understanding both how this could be exploited as well as when and how ScreenOS started using Dual EC required a closer, forensic investigation.

Towards those ends, our team acquired several Juniper Secure Services Gateway (SSG) devices running ScreenOS, including an Intel x86–based SSG-550M and an ARM-based SSG-5. Additionally, we acquired ScreenOS firmware binaries for 50 different point releases of ScreenOS across the two architectures and five major versions. Table 7 summarizes the versions we examined. We verified the MD5 or SHA-1 hash of each firmware version against those published on Juniper's website — except for the 6.0 revisions for which we could find no published hashes.

Each revision of the firmware contains a modified copy of the "engine" variant of OpenSSL 0.9.6c. Identifying OpenSSL's functions in a binary is particularly easy. One need only look up error and function ordinals used as arguments to error handling functions in lists of error codes included in the OpenSSL

[xi] ScreenOS 6.3's cascade is identical.

[xii] https://gist.github.com/hdm/0fbaf7408a6c7e0566c5

[xiii] Adam Langley documented some of this effort as it appeared on Twitter https://www.imperialviolet.org/2015/12/19/juniper.html.

Table 7: ScreenOS firmware versions we examined.

Device series	Architecture	Version	Revisions
SSG-500	Intel x86	6.3.0	12b
SSG-5/SSG-20	ARM-BE	5.4.0	1–3, 3a, 4-16
		6.0.0	1–5, 5a, 6–8, 8a
		6.1.0	1–7
		6.2.0	1–8, 19
		6.3.0	1–6

source code. Once the OpenSSL functions were identified, we identified the pseudorandom number generator functions and, ultimately, how those were used to construct nonces and Diffie–Hellman keys in IKE.

Extracting the strings from each of the firmware binaries revealed the presence of an OpenSSL elliptic curve component in versions 6.2.0r1 and later which suggested that that was where Dual EC was introduced. Nevertheless, we examined all of the five major versions for which we had firmware. For each of those major versions, we reverse-engineered the earliest revision we had, the latest revision we had, and one or more intermediate revisions to determine how the IKE nonce generation — including the pseudorandom number generator used — changed over time.

Listing 4.1 shows the decompiled source code for the ScreenOS PRG version 6.2.0r1. Note that identifiers such as function and variable names are not present in the binary; we assigned these names based on analysis of the apparent function of each symbol. Similarly, specific control flow constructs are not preserved by the compilation/decompilation process. For instance, the `for` loop on line Line 22 may in fact be a `while` loop or some other construct in the actual Juniper source. Decompilation does, however, preserve the functionality of the original code. For clarity, we have omitted FIPS checks that ensure that the ANSI X9.31 generator [87] has not generated duplicate output.

Superficially, the ScreenOS implementation appears consistent with Juniper's description: When `prng_generate()` is called, it first potentially reseeds the X9.31 PRG state (Lines 18–20) via `prng_reseed()`. When

[87] Accredited Standards Committee (ASC) X9, Financial Services. *ANSI X9.31-1998: Digital Signatures using Reversible Algorithms for the Financial Services Industry (rDSA)*. 1998

Listing 4.1: The Core ScreenOS 6.2 PRG Subroutines.

```
1  void prng_reseed(void) {
2    blocks_generated_since_reseed = 0;
3    if (dualec_generate(prng_temporary, 32) != 32)
4      error_handler("FIPS ERROR: PRNG failure, "
5                    "unable to reseed\n", 11);
6    memcpy(prng_seed, prng_temporary, 8);
7    prng_output_index = 8;
8    memcpy(prng_key,
9           &prng_temporary[prng_output_index], 24);
10   prng_output_index = 32;
11 }
12
13 void prng_generate(void) {
14   int time[2];
15   time[0] = 0;
16   time[1] = get_cycles();
17   prng_output_index = 0;
18   ++blocks_generated_since_reseed;
19   if (!one_stage_rng())
20     prng_reseed();
21   for (; prng_output_index <= 31;
22        prng_output_index += 8) {
23     // FIPS checks removed for clarity
24     x9_31_generate_block(time, prng_seed, prng_key,
25                          prng_block);
26     // FIPS checks removed for clarity
27     memcpy(&prng_temporary[prng_output_index],
28            prng_block, 8);
29   }
30 }
```

`prng_reseed()` is called, it invokes the Dual EC PRG to fill the 32-byte buffer `prng_temporary`. From this buffer, it extracts a seed and cipher key for the ANSI X9.31 generator. Once the X9.31 PRG state is seeded, the implementation then generates 8 bytes of X9.31 PRG output at a time (Line 25) into `prng_temporary`, looping until it has generated 32 bytes of output (Lines 22–29), using the global variable `prng_seed` to store the ANSI X9.31 seed state, updating it with every invocation of `prng_generate_block()`.

However, upon closer inspection, the behavior of the generator is subtly different. This is due to two coupled issues: First, `prng_reseed()` and `prng_generate_blocks()` share the static buffer

[xiv] The global variable reuse was first publicly noted by Willem Pinckaers on Twitter. Online: https://twitter.com/_dvorak_/status/679109591708205056, retrieved February 18, 2016.

`prng_temporary`; and second, when `prng_reseed()` is invoked, it fills `prng_temporary` (Line 3) and then sets the static variable `prng_output_index` to 32 (the size of the Dual EC output).[xiv] Unfortunately, `prng_output_index` is *also* the control variable for the loop that invokes the ANSI X9.31 PRG in `prng_generate()` at Line 22. The consequence is that whenever the PRG is reseeded, `prng_output_index` is 32 at the start of the loop and therefore no calls to the ANSI X9.31 PRG are executed. Thus, Dual EC output is emitted directly from the `prng_generate()` function.

In the default configuration, `one_stage_rng()` always returns false so X9.31 is reseeded on *every* call. There is an undocumented ScreenOS command, `set key one-stage-rng`, which is described by a string in the command-parsing data-structure as "Reduce PRG to single stage." Invoking this command effectively disables reseeding until this setting is changed.

When combined with the cascade bug described above, disabling reseeding introduces a different security vulnerability: The first block emitted after reseed is precisely the concatenation of the 8-byte seed and the 24-byte key used for future blocks of output from the ANSI X9.31 PRG. An attacker who is lucky enough to observe an immediate post-reseed output can predict the rest of the PRG stream until the next reseed *even without knowing* $\log_P Q$.[xv]

Had `prng_output_index` not been used in `prng_reseed`, the reuse of `prng_temporary` would be safe.

As described in Section 4.9, the index variable used in the for loop in `prng_generate` changed from a local variable to the `prng_output_index` global variable between the final version of ScreenOS 6.1 and the first version of 6.2.

4.5 Interaction with IKE

As suggested by the exploit description, the primary concern with a Dual EC implementation is that an attacker may be able to use public information emitted by the PRG to extract the Dual EC internal

[xv] There are technical obstacles to overcome. X9.31 uses the current time (parameter DT in the specification; implemented as the processor cycle counter in ScreenOS) as an input to the PRG. As long as the time value can be guessed (or brute forced), the X9.31 generator's output can be predicted. As `one-stage-rng` is off by default and the command that enables is is undocumented, we did not study this issue in depth.

state, and use this to predict future secret values. Because ScreenOS is not only a firewall but also a VPN device, the natural target is Internet Key Exchange (IKE) [59,61], the key establishment protocol used for Internet Protocol Security (IPsec) [88]. Note that the existence of a Dual EC generator does not by itself imply that Juniper's IKE implementation is itself exploitable, even in conditions where the attacker knows the Dual EC discrete log. There are a number of parameters that affect both the feasibility and cost of such an attack.

4.5.1 Overview of IKE

IKE (and its successor IKEv2) is a traditional Diffie–Hellman-based handshake protocol in which two endpoints (dubbed the *initiator* and the *responder*) establish a *Security Association* (SA) consisting of parameters and a set of traffic keys which can be used for encrypting traffic. Somewhat unusually, IKE consists of two phases:

- *Phase 1 (IKEv1)/Initial Exchange (IKEv2)*: Used to establish an "IKE SA" which is tied to the endpoints but not to any particular class of non-IKE network traffic.
- *Phase 2 (IKEv1)/CREATE_CHILD_SA (IKEv2)*: Used to establish SAs which protect non-IKE traffic (typically IPsec). The IKE messages for this phase are protected with keys established in the first phase. This phase may be run multiple times with the same phase 1 SA in order to establish multiple SAs (e.g., for different IP host/port pairs), but as a practical matter many VPN connections compute only one child SA and use it for all traffic.

For simplicity, we will use the IKEv1 terminology of phase 1/phase 2 in the rest of this document.

IKE messages are composed of a series of "payloads" such as KE (key exchange), Ni (initiator nonce), and Nr (responder nonce).

The first IKE phase consists of a Diffie–Hellman exchange in which both sides exchange Diffie–Hellman shares and a nonce, which are combined to form the derived keys. The endpoints may be authenticated in a variety of ways including a signing key and a statically configured shared secret. The second IKE

[88] S Kent and K Seo. *Security Architecture for the Internet Protocol.* RFC 4301. Internet Engineering Task Force, Dec. 2005

phase may involve a DH exchange but may also just consist of an exchange of nonces, in which case the child SA keys are derived from the shared secret established in the first phase.

At this point, we have a conceptual overview of how to attack IKE where ScreenOS is the responder: (1) using the responder nonce in the first phase, compute the Dual EC state; (2) predict the responder's DH private key and use that to compute the DH shared secret for the IKE SA; (3) using the keys derived from the IKE SA, decrypt the second phase traffic to recover the peers' nonces and public keys (in the best case, the responder nonce and private key can be computed by running Dual EC forward; otherwise one can repeat the Dual EC attack); and (4) use those to compute the shared secret for the second phase SA and thereby the traffic keys. Use those keys to decrypt the VPN traffic.

However, while this is straightforward in principle, there are a number of practical complexities and potential implementation decisions which could make this attack easier or more difficult (or even impractical) as described below.

4.5.2 Nonce Size

The first question we must examine is whether the attacker ever gets a complete Dual EC block. As Checkoway et al. [21] describe in detail, it is only practical to exploit Dual EC if provided with nearly a complete point output. As specified, Dual EC emits only 30 bytes of the 32-byte point, which requires the attacker to try approximately half of the remaining 2^{16} values to find the state, and the work factor goes up exponentially with the number of missing bytes, so exploitation rapidly becomes impractical the less of the point the attacker has.

Many reasonable implementation strategies could result in an attacker obtaining only small fractions of a point. For example, unlike TLS, IKE has a variable-length nonce, which is required to be between 8 and 256 bytes in length [59]. If a nonce length below 30 bytes were used, it could significantly increase the amount of work required to recover the Dual EC state

However, as of version 6.2 ScreenOS uses a 32-byte nonce made from two successive invocations of Dual EC, with the first supplying 30 bytes and the second supplying 2 bytes. As described in Section 4.2,

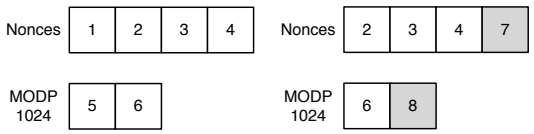
this is nearly ideal from the perspective of the attacker because it can use the first 30 bytes (the majority of the point) to determine possible states, and then narrow the results by checking which states produce the correct value for the remaining two bytes. In practice, this usually results in 1 to 3 possible states.

4.5.3 Nonces and DH Keys

Although the IKE messages contain both a nonce and a DH share our analysis of Juniper's IKE implementation indicates that the KE payload containing the DH share is encoded *before* the Nr (nonce) payload. If (as is natural), the keys and nonces are generated in the same order as they are encoded, then it will not be possible to use Nr from one connection to attack that same connection. This is because Dual EC state recovery only allows you to predict future values, not recover past values. While not necessarily fatal to the attacker, because nonces generated in one connection might be used to predict the DH private keys generated in some subsequent connection; this would not be ideal from the attacker's perspective, especially if connection establishment is infrequent.

Conveniently for the attacker, however, ScreenOS also contains a pre-generation feature that maintains a pool of nonces and DH keys which can then be used in new IKE connections rather than generating them in the critical path (i.e., during the handshake). The pooling mechanism is quite intricate and appears to be designed to ensure that enough keys are always available while avoiding consuming too much run time on the device.

Summarized briefly, independent FIFO queues are maintained for nonces, each finite field DH group (MODP 768, MODP 1024, MODP 1536, and MODP 2048), and (in version 6.3) each elliptic curve group (ECP 256 and ECP 384). The sizes of these queues depend on the number of VPN configurations which have been enabled for any given group. For instance, if a single configuration is enabled for a group then that group will have queue size of 2 and disabled groups have a queue size of 0. The size of the nonce queue is set to be twice the aggregate size of all of the DH queues. So, for instance, if only the MODP 1024 group is configured, then the initial queue size will be (MODP $1024 = 2$, nonce $= 4$). Or, if two VPN configurations are set to use MODP 1024 and one configuration is set to use MODP 1536, initial queue size will be (MODP $1024 = 4$, MODP $1536 = 2$, nonce $= 12$). At initial startup time,



(a) At system startup. **(b)** After a DH exchange.

Figure 6: Nonce queue behavior during an IKE handshake. Numbers indicate generation order, and values generated after the handshake are shaded. During a DH exchange, outputs 1 and 5 are used as the nonce and key, advancing the queue, and new outputs are generated to fill the end of the queue.

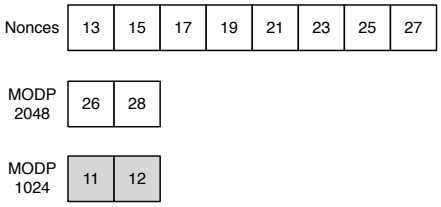


Figure 7: Queue state after 8 MODP 2048 exchanges. Numbers indicate generation order, and stale values are shaded. If several connections have been made to the same DH group, the other DH group can grow stale as all nonces that were generated before those keys are used up.

the system completely fills all the queues to capacity and then sets a timer that fires every second to refill the queues if any values have been used.[xvi] If a nonce or a DH key is ever requested when the queue is empty, then a fresh value is generated on the fly.

Importantly, the queues are filled in priority order with nonces being the highest priority followed by the groups in descending order of cryptographic strength (ECP 384 down to MODP 768). This means that in many (but not all) cases, the nonce for a given connection will precede the keys for that connection in the random number sequence.

Figure 6 shows a (somewhat idealized) sequence of generated values,[xvii] with the numbers indicating the order in which they were generated before and after an IKE DH exchange. Figure 6a shows the situation after startup: The first four values are used to fill the nonce queue and the next two values are used to generate the DH shares. Thus, when the exchange happens, it uses value 1 for the nonce and value 5 for the key, allowing the attacker to derive the Dual EC state from value 1 and then compute forward to find the DH private key. After a single DH exchange, which requires one DH key and one nonce, the state is as shown in Figure 6b, with the new values shaded. Note that the next-in-line values continue to have the property that the nonce was generated before the DH share. Because nonce computation is

[xvi] Note: only one value is generated per second, so if several values are used, it takes some time to refill the queue.

[xvii] For simplicity, we represent multiple consecutive invocations of the PRG as a single value and ignore invocations of the PRG for non-IKE purposes. In addition, because the queues are refilled asynchronously with respect to the IKE exchanges, there is a race condition between values being consumed and being refreshed. The pattern shown here and below is the result of assuming that the timer fires between handshakes. If it fires more frequently (i.e., between each DH and nonce encoding), then the nonces become even and the DH shares become odd. Mixed patterns are also possible.

prioritized over key generation, in this simple configuration where you have a single DH group that is used for every handshake, then as long as handshakes are done reasonably slowly (giving the background task enough time to fill the queue) the nonce used for a given handshake will always have been generated prior to the DH key for that handshake. Of course, if a large number exchanges are run in succession (i.e., outpacing the background task) it is possible to exhaust both queues entirely, at which point the request for a key or nonce will cause the value to be generated immediately, resulting in the DH key being computed before the nonce.

4.5.4 Non-DH Phase 2 Exchanges

As noted above, the phase 2 exchange need not include a new DH exchange; implementations can simply do a nonce exchange and generate fresh keys (although Juniper's documentation recommends doing DH for phase 2 as well [89]). In this case, IKE will consume an additional nonce from the nonce queue but not a new DH key from the DH key queue. In the case where endpoints do a single phase 1 exchange and then a phase 2 exchange, with only the former doing DH, then setting up a VPN connection setup consumes two nonces and one DH key. However, because the nonce queue is twice as large as the DH queue, as long as the refill timer fires reasonably often with respect to the handshakes it is not possible to exhaust the nonce queue (thus causing a fresh PRG value to be generated) while there is still a stale DH value in the DH queue. Note that if the nonce and DH queues were the same size, then non-DH phase 2 exchanges would frequently cause keys to be stale with respect to the nonce.

In addition, if *multiple* non-DH phase 2 exchanges are done within a single phase 1 exchange, then it is possible to empty the nonce queue while there are still values in the DH queue. In this case, it will only be possible to decrypt connections established using those values if the attacker has recorded previous nonces, rather than decrypting a connection in isolation. Similarly, the current nonces could be used to decrypt future connections but not the connections they are transmitted with.

[89] Juniper Networks. *Concepts & Examples ScreenOS Reference Guide: Virtual Private Networks*. Rev. 02, Dec. 2012.

4.5.5 Multiple Groups

If ScreenOS is configured to use multiple groups, then it is possible to have the shares for one group become stale with respect to the nonces queue, as shown in Figure 7, which shows the result of eight MODP 2048 exchanges on the queues. The shaded MODP 1024 values were all generated before any of the remaining nonces. If the attacker starts listening at this point and observes a MODP 1024 exchange, he will not be able to decrypt it.

4.5.6 Recovering traffic keys

As described above, IKE comes in two versions (IKEv1 and IKEv2) which are slightly different. Furthermore, each version uses a somewhat unusual two-phase approach to protecting traffic. In this section, we describe the phases and the authentication modes that determine whether or not protected traffic can be passively decrypted.

IKEv1, phase 1 IKEv1 defines four authentication modes for phase 1: digital signatures, two modes using public-key encryption, and preshared keys [59]. Although the details vary, each mode computes a shared secret, SKEYID, derived from secret values (e.g., nonces and Diffie–Hellman keys) exchanged in the handshake. Next, the authentication keying material, SKEYID_a, and encryption keying material, SKEYID_e, are derived from SKEYID. Finally, the traffic keys used to protect phase 2 are derived from SKEYID_e in an algorithm-specific manner. We omit further discussion of authentication key material below as a passive adversary does not need it. An active adversary wishing to tamper with the traffic would need to use SKEYID_a in the straightforward manner prescribed by the RFC.

- **Authentication with digital signatures.** In this mode, the initiator and responder nonces and DH public keys are exchanged in the clear. Starting with the responder's nonce, an attacker who can recover the responder's DH private key has all of the material necessary to compute SKEYID and thus traffic keys.
- **Authentication with public key encryption.** IKEv1 defines two public-key encryption modes for authentication. The revised mode uses half the number of public-key encryptions and decryptions the other mode uses, but are otherwise similar. In these modes, the Diffie–Hellman public keys

are exchanged in the clear but each peer encrypts its nonce using the other's public key. These modes require the initiator to know the responder's public key prior to the handshake. Each peer decrypts the other's nonce and computes SKEYID. Since nonces are encrypted, even if an attacker can recover the responder's nonce (e.g., by capturing a nonce in the clear from a previous connection, recovering the Dual EC state, and walking the generator forward), the initiator's nonce is also encrypted, thus stopping the attack.

- **Authentication with preshared keys.** In this mode, a preshared key needs to be established out of band. The DH public keys and nonces are exchanged in the clear. The encryption keying material, SKEYID_e is derived from the preshared key, the nonces, and the DH keys. An attacker who can recover a DH private key can perform an offline attack on the preshared key. Depending on the strength of the PSK, this offline attack may be trivial or may be computationally intractable.[xviii]

IKEv1, phase 2 After phase 1 completes, there is a second phase, called Quick Mode, which involves another exchange of nonces and, optionally, another Diffie–Hellman exchange for forward secrecy [61]. As the messages for phase 2 are protected by the keys established during phase 1, there is no additional encryption. Thus, an attacker who has successfully recovered the phase 1 keys can decrypt phase 2 messages. At this point, if another Diffie–Hellman key exchange is used, the attacker can either run the Dual EC-state-recovery attack again or simply walk the Dual EC generator forward to recover the Diffie–Hellman private key. If only nonces are exchanged, then no additional work is necessary. In either case, the attacker can compute the traffic keys and recover plain text.

IKEv2, phase 1 A connection in IKEv2 begins by exchanging two request/response pairs which form the initial exchange. The first pair of messages, called IKE_SA_INIT, exchange DH public keys and nonces in the clear. The peers use these to compute a shared secret, SKEYSEED, from which all traffic keys are derived. These keys are used to protect the following messages.

[xviii] Anecdotally, the preshared keys used in practice are often quite weak. For example, FlyVPN's "How To Setup L2TP VPN On Android 4" instructs the user to "Input 'vpnserver' letters into 'IPSec pre-shared key.' " https://www.flyvpn.com/How-To-Setup-L2TP-VPN-On-Android-4.html, retrieved February 18, 2016.

This first exchange contains all of the information necessary for the attacker to recover the Dual EC state and compute a DH private key and thus derive SKEYSEED. This stands in contrast to IKEv1 where the authentication mode influences key derivation and hence, exploitability.

The second exchange, called IKE_AUTH, is encrypted using keys derived from SKEYSEED and is used to authenticate each peer, but plays no role in decryption. At this point, a child security association (CHILD_SA) is set up which can be used for protecting VPN traffic.

IKEv2, phase 2 IKEv2 does contain a second phase, called CREATE_CHILD_SA, which can be used to create additional child security associations. One use of this phase is periodic rekeying. The use of a second phase is optional.[xix]

Similar to IKEv1's second phase, nonces and, optionally, Diffie–Hellman public keys are exchanged. As before, when DH keys are used, an attacker may either perform the attack to recover Dual EC's state a second time or walk the generator forward.

4.6 Attacking IKE

To validate the attacks we describe above, we purchased a Juniper Secure Services Gateway 550M VPN device, and modified the firmware version 6.3.0r12 in a manner similar to the 2012 attack. This required us to generate a point Q for which we know the trapdoor $(\log_P Q)^{-1}$, and to modify the Dual EC Known Answer Test (KAT) correspondingly. To install the firmware on the device, we further modified a non-cryptographic checksum contained within the header of the firmware.[xx]

Using the new firmware, we next configured the device with three separate VPN gateways: (1) configured for IKEv1 with a PSK, (2) configured for IKEv1 with a 1024-bit RSA signing certificate, and (3) configured for IKEv2 with a PSK. For each configuration, we initiated VPN connections to the box using strongSwan [90]. By capturing the resulting traffic, we were then able to extract the nonces in the

[xix] "The second request/response (IKE_AUTH) transmits identities, proves knowledge of the secrets corresponding to the two identities, and sets up an SA for the first (and often only) AH and/or ESP CHILD_SA" [61].

[xx] If a code-signing certificate is installed on the device, firmware updates require the presence of a valid digital signature on the new firmware using the key in that certificate. Since we did not have a certificate installed, we were able to omit this signature.

[90] StrongSwan. *strongSwan: the OpenSource IPsec-based VPN Solution*. Nov. 2015

IKE handshakes and run the Dual EC attack to recover the state of the random number generator for each connection. As previously discussed, since the 32-byte nonces consist of the concatenation of two consecutive 30-byte Dual EC blocks, truncated to 32-bytes, we used the first 30 bytes of the nonce to recover a potential state value, and then confirmed this guess against the remaining 2 bytes of the nonce.

From this point, we generated a series of Dual EC outputs to obtain a private exponent consistent with the Diffie–Hellman public key observed in the traffic. This required a single modular exponentiation per potential exponent x, followed by a comparison to the extracted key exchange payload value. Given the correct private exponent, we then obtained the shared secret from the initiator value, thereby determining the DH shared secret g^{xy}. Given the Diffie–Hellman shared secret, we implemented the remaining elements of the IKEv1 and IKEv2 standards [59,61] in order to calculate the Phase 1 (Aggressive Mode) keying material (for IKEv1) and the corresponding IKE_SA_INIT/IKE_AUTH keying material (for IKEv2). This information encrypts the subsequent handshake messages, and is itself used to calculate the key material for subsequent payloads, including Encapsulated Secure Payload (ESP) messages. A challenge in the IKEv1 PSK implementation is the need to incorporate an unknown PSK value into the PRF used to calculate the resulting key material. For our proof of concept implementation we used a known PSK, however without knowledge of this value, an additional brute-force or dictionary attack step would have been required. No such problem exists for the IKEv1 certificate connections, or for IKEv2 PSK.

Using the recovered key material, we next decrypt the remaining traffic, which in each case embeds a second Diffie–Hellman handshake with additional nonces and Diffie–Hellman ephemeral public keys. Since this handshake is also produced from the same generator, we can simply wind the generator forward (or restart with a nonce drawn from the second phase handshake) to recover the corresponding Diffie–Hellman private keys. This new shared secret can then be used to calculate the resulting key material. All subsequent traffic that we see and decrypt utilized the Encapsulating Security Payload (ESP) protocol [91] in tunnel mode.

[91] S Kent. *IP Encapsulating Security Payload (ESP)*. RFC 4303. Internet Engineering Task Force, Nov. 2005

4.7 Passively detecting ScreenOS

An adversary who knows the Dual EC Q parameter — either Juniper's 2008 point or the unauthorized 2012 point — may wish to detect vulnerable versions of ScreenOS by passively watching network traffic. In theory, such an adversary has several avenues open to it. The easiest approach is to attempt the attack on every VPN connection to see if the attack is successful. Alternatively, the adversary could attempt to fingerprint VPN boxes and only perform the attack on connections that match.

Dual EC is known to have a small, but non-negligible, bias. In particular, Schoenmakers and Sidorenko[77] and[78] give a procedure to distinguish 30-byte blocks generated uniformly at random from those generated by Dual EC. The basic idea is to count how many points on the curve have x-coordinates that agree with the 30-byte block in their least significant 30 bytes. In both the uniformly at random case and the Dual EC case, the number of points on the curve that match follow a normal distribution. In order to use the this distinguisher, one needs to see a sufficient number of 30-byte blocks (in the form of IKE nonces) to state with high confidence that the blocks came from one distribution or the other.

We empirically computed the parameters of these two distributions to see how difficult this task is. We generated 2 million 30-byte blocks using Dual EC and an additional 2 million blocks uniformly at random and performed the point counting. We estimate the distributions' parameters by fitting the data using maximum likelihood estimation. The results are not encouraging. When generated uniformly at random, the number of points on the curve that agree with the generated block have parameters $\mu = 65536.02$ and $\sigma = 256.05$. When generated using Dual EC, the parameters are $\mu = 65536.78$ and $\sigma = 256.06$. This approach is unlikely to work without seeing tens or hundreds of thousands of connections.

Although detecting the Dual EC bias requires a massive number of connections, the adversary's task is actually much easier due to a bug in ScreenOS's Dual EC implementation. ScreenOS contains a customized version of OpenSSL and uses OpenSSL's elliptic curve and arbitrary-precision (BIGNUM) routines to implement Dual EC. The OpenSSL function to convert a BIGNUM to an array of bytes is

[xxi] This defect was corrected quite recently, years after the version of OpenSSL ScreenOS uses was written. https://mta.openssl.org/pipermail/openssl-commits/2016-February/003520.html

`BN_bn2bin()`. Due to a design defect in OpenSSL's API, there is no way to correctly use `BN_bn2bin` without first determining how many bytes it will use — using `BN_num_bytes()` — and zero-padding, and only then using `BN_bn2bin()`:[xxi]

```
int size = BN_num_bytes(x);
memset(buffer, 0, 30 - size);
BN_bn2bin(x, buffer + (30 - size));
```

ScreenOS's Dual EC implementation omits the zero padding when converting from BIGNUMs to binary output. The upshot is that neither the first nor the thirty-first byte of a nonce will ever be zero.

Thus, if the adversary ever sees a zero byte in either position, it can conclude that the implementation is not Juniper's Dual EC.

If the nonces are generated uniformly at random, then we expect each of these bytes to be zero with (independent) probability 1/256. Thus, the probability that after n nonces without zeros in those positions, the nonce was generated uniformly at random is $(255/256)^{2n}$.

This bug does not affect the exploitability of the Dual EC generator; however, it can lead to a few additional potential internal states to check as described in Section 4.2.

4.8 Counting ScreenOS Devices

The zero-detection technique described in the previous section also suggests a method of counting the number on the Internet running vulnerable versions ScreenOS, as discussed in this section. As a convenience, in the discussion below we say a host *sent a zero* to mean that during an IKEv1 or IKEv2 key exchange, the responder nonce contained a zero at position 0 or 30.

Data collection

We used Internet-wide scans to attempt to measure the population of affected devices. Because the signal to be measured is fairly small, we performed the scan in multiple passes, starting by identifying all hosts which might potentially be running an affected version of ScreenOS and then repeatedly scanning them,

dropping each one as soon as it sent a zero, leaving us with a final group of hosts which are highly probable to be ScreenOS. The data collection was performed during April 2016 using ZGrab, an application-layer scanner that operates with ZMap [92].

In the first phase, we performed an Internet-wide scan of all IPv4 addresses on port 500 to determine which hosts were configured to negotiate IKEv1 and IKEv2. Next, we used a custom ZGrab module, which acted as an IKE initiator and attempted to perform the first key exchange phase for both IKEv1 and IKEv2. Our initiator client offered a set of cipher suites chosen from the most commonly supported options observed by Adrian et al. [93].We tested this packet[xxii] on the default configuration of our vulnerable NetScreen device to verify that it was accepted.

Our initial scan found 7,703,858 hosts that responded to our probes with valid IKE packets, indicating that they support IKEv1 or IKEv2. Of those hosts, only 2,263,314 were willing to perform an initial key exchange with us. The remaining were likely configured to only negotiate with specific whitelisted IPs. Because the affected versions of ScreenOS send 32-byte nonces, we excluded all hosts responding with nonces of other lengths. This left us with 343,467 hosts: 94,201 which just accepted IKEv1, 131,080 which just accepted IKEv2, and 118,186 which accepted both. If a host responded to both IKEv1 and IKEv2, it was subsequently scanned only for IKEv2 in an effort to minimize our impact; see the limitations section below for a discussion of the consequences of this choice.

In the second phase, we periodically scanned all of the remaining hosts that had responded in the preceding scan with a 32-byte nonce without sending a zero. This phase consisted of 601 rounds of scanning, spread out over five days to minimize the impact of our scans on the hosts. By the end of the second phase, 58,695 IKEv1 hosts and 71,360 IKEv2 hosts had stopped responding and were not scanned again. Of the total remaining $N = 213{,}412$ hosts, 31,801 IKEv1 and 176,619 IKEv2 hosts sent a zero, leaving 3,705 IKEv1 hosts and 1,287 IKEv2 hosts that responded to 602 scans (the first phase, baseline scan and the 601 second phase scans) without sending a zero.

[xxii] We used the 3DES-CBC-SHA, AES-CBC-SHA, AES-CBC-SHA, 3DES-CBC-MD5, DES-CBC-MD5, DES-CBC-SHA, AES-CBC-MD5, and AES-CBC-MD5 cipher suites.

[92] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. *USENIX Security*, 2013.
[93] David Adrian et al. *ACM CSS*, 2015.

If we assume that non-ScreenOS hosts construct nonces uniformly at random — this does not appear to be actually true, as discussed below — the probability that a non-ScreenOS host sent n nonces without sending a zero is $p_n = (255/256)^{2n}$. In particular, the probability that a non-ScreenOS host sent 602 nonces without sending a zero is $p_{602} \approx .90\%$. If S of the N total hosts run ScreenOS, we would expect to see $S + (N - S) \cdot p_{602}$ hosts that do not send a zero after 602 rounds. We measured $3705 + 1287 = 4992$ hosts that did not send a zero and so we estimate that there are about $S \approx 3103$ ScreenOS hosts in our data set. Using data from Censys [94] scans of other ports running HTTPS, SSH, and/or Telnet management services, we were able to confirm that 447 of these hosts were in fact running some version of ScreenOS.

Limitations

There are several limitations to the data and analyses discussed above: hosts are a weak proxy for traffic, nonpublic hosts, IP blocking, scanning methodology, and analysis methodology. We discuss each in turn.

Fundamentally, we would like to understand how much VPN traffic is vulnerable to Juniper's "knowledgeable attacker." Unfortunately, a network scan cannot answer that question. Instead, we can only compute a weak proxy for this number by determining how many vulnerable VPN devices are on the Internet.

A second potentially-confounding factor is that not all VPN devices are publicly accessible. These devices may be configured to create VPN connections between several private networks and thus would refuse our baseline scan handshake if peer addresses had been explicitly configured in the host. This stands in contrast to a scan to detect vulnerabilities in public web servers using TLS and represents a fundamental limitation of network mass scans for understanding VPN infrastructure.

Third, our measurements saw a significant drop in response rates over time. Some of this may be due to natural movement of hosts over time, as we attempted to minimize impact by again scanning hosts that had previously responded successfully. In some cases, despite our attempts to scan slowly and over a

[94] Zakir Durumeric et al. *ACM CCS*, 2015.

long period of time, we may have inadvertently triggered ScreenOS's built-in anti-DoS protection for some hosts. We are unable to distinguish these cases.

Fourth, our scanning methodology turned out to be based on flawed assumptions, namely that non-ScreenOS hosts would send uniformly random nonces and that IKEv1 and IKEv2 behavior would be the same. A frequency analysis of the bytes in various positions in the nonces we collected suggests that the bytes are not uniformly distributed in non-ScreenOS hosts (i.e., even nonces from hosts which send zeros are not uniform) and that, for some reason, the nonuniformity is stronger in IKEv2 hosts, making those results even less reliable. Thus, our decision to scan hosts that support both IKEv1 and IKEv2 only using IKEv2 was an error.

4.9 Discussion

Much attention has been paid to the 2012 compromise of Juniper's ScreenOS source code by unknown parties. In this chapter we have shown that the vulnerabilities announced by Juniper can be traced largely to the pre-existing design of Juniper's ScreenOS random number generator. Specifically, we argue that

Listing 4.2: The Core ScreenOS 6.1 PRG Subroutine.

```
1  void prng_generate(char *output) {
2    unsigned int index = 0;
3    int time[2];
4
5    // FIPS checks removed for clarity
6    if (blocks_generated_since_reseed++ > 9999)
7      prng_reseed();
8    // FIPS checks removed for clarity
9    time[0] = 0;
10   time[1] = get_cycles();
11   do {
12     // FIPS checks removed for clarity
13     prng_generate_block(time, prng_seed, prng_key,
14                         prng_block);
15     // FIPS checks removed for clarity
16     memcpy(&output[index], prng_output_block,
17            min(20-index, 8));
18     index += min(20-index, 8);
19   } while (index <= 19);
20 }
```

Juniper's design is exploitable due to a series of deliberate design decisions, accidents, and oversights on the part of the ScreenOS developers.

Below we review each of the conditions that are required to produce an exploitable PRG in the Juniper system:

1. **Implementation of Dual EC.** The cascade design of Juniper's double PRG, which employs Dual EC to seed the PRG on each call seems a surprising choice, given the performance limitations of Dual EC. Notably, the transition from ScreenOS 6.1 (X9.31 only) to 6.2 (Dual EC and X9.31) involved the addition of a *nonce pre-generation* queue to the existing DH key queues.[xxiii] One potential motivation for this change could be the additional security assurance provided by Dual EC. However, we note that Juniper did not seek FIPS certification of the Dual EC generator, despite the fact that following the deprecation of the ANSI X9.31 generator on January 1, 2016, it would have been the only FIPS-certified PRG in their product.[xxiv]
2. **Presence of a Dual EC/ANSI cascade flaw.** Even with Dual EC present in the ScreenOS devices, the use of a cascade between Dual EC and ScreenOS should have prevented the known state recovery attacks. As detailed in Section 4.4, this protection is not available due to flaws in the cascade implementation, which allows for the exfiltration of unprocessed Dual EC output. This flaw is particularly perplexing. Compare the `prng_generate()` function in version 6.1 (Listing 4.2) with the analogous function in version 6.2 (Listing 4.1). Apart from minor changes arising from moving from generating 20 bytes at a time to generating 32 bytes at a time and always reseeding rather than reseeding based on a counter, the functions look quite similar. However, for some reason, the loop index variable was changed from a local variable to a global variable.
3. **Always reseeding.** In version 6.1, ScreenOS reseeded the X9.31 PRG from system entropy every 10,000 calls (hard-coded; see Listing 4.2). However, in version 6.2, the reseeding mechanism was repurposed to produce the cascade by always reseeding. When combined with the cascade flaw

[xxiii] To give a rough estimate of the performance difference, we implemented Dual EC and ANSI X9.31 using the same procedure used in ScreenOS and measured how long it takes to generate 32-byte blocks. Dual EC takes roughly 125 times as long as X9.31.

[xxiv] A review of the Cryptographic Module Validation Program (CMVP) certification lists [62] shows that all ScreenOS FIPS certification certificates have indeed been de-listed as of February 2016.

Table 8: ScreenOS Features by Version.

Version	PRG	Reseed period	Reseed bug	DH queue	Nonce queue	Nonce size	DH groups supported
6.1.0	X9.31	10000		✓		20 bytes	MODP 768, 1024, 1536, 2048
6.2.0	Dual EC + X9.31	1	✓	✓	✓	32 bytes	MODP 768, 1024, 1536, 2048
6.3.0	Dual EC + X9.31	1	✓	✓	✓	32 bytes	MODP 768, 1024, 1536, 2048 ECP 256, 384

Between versions 6.1.0 and 6.2.0, a cluster of changes were made to the PRG and IKE subsystems. In the PRG subsystem, the switch to (1) Dual EC + X9.31; (2) reseeding on every call; and (3) the bug in reseed that causes X9.31 to be skipped produce the necessary conditions to attack IKE. In the IKE subsystem, changing the nonce size from 20 bytes to 32 bytes moves the attack from completely impractical to nearly best-case scenario, from an attacker's point of view. The introduction of a nonce queue changes the nature of the attack such that, in the usual case, an attacker can decrypt a session based solely on that session's traffic.

Version 6.3.0 is nearly identical to 6.2.0 but supports elliptic curve Diffie–Hellman groups. In contrast to the changes between 6.1.0 and 6.2.0, this may actually make an attacker's job harder; see Section 4.5.5.

described above, all PRG output comes from Dual EC, increasing the probability that a specific value observed by the attacker can be used to recover PRG state.

4. **Use of 32-byte IKE nonces.** The IKE standards do not provide a specific recommendation for nonce length, stating only that nonces should be between 8 and 256 bytes, and that nonces should be at least half the key size of the PRF used. The last version of ScreenOS without Dual EC was 6.1.0r7 and specified 20 byte nonces. In the subsequent release, ScreenOS 6.2.0r1, Juniper developers added Dual EC and modified the IKE nonce size from 20 to 32 bytes. Efficiently recovering the state of the Dual EC generator requires at a minimum 26 bytes of unprocessed PRG output, and as discussed in Section 4.5.2, having greater than 30 bytes expedites the state recovery attack.

5. **Modifying the order of nonce and key generation.** The ScreenOS IKEv1 and IKEv2 implementations both output IKE Key Exchange prior to the IKE Nonce packet. However, this output order does not reflect the *generation* order of the same values in all versions of ScreenOS. In particular, the addition of *nonce queues* in ScreenOS 6.2.0r1 effectively guarantees that in most cases a non-loaded system will generate a nonce immediately prior to the Diffie–Hellman private key that will be used in a given handshake. In practice, this facilitates state recovery attacks that can recover secret keys (and thus enable decryption) within a single IKE handshake, significantly improving the effectiveness of passive attacks.

All told, in the course of a single version revision, Juniper made a series of changes that combined to produce a system which only required the attacker to know the discrete log of Q to be exploitable. See Table 8 for a summary of changes. For a randomly selected Q, or a point chosen using the nothing-up-my-sleeve process proposed in ANSI X9.82 [95], calculating d is likely to be infeasible. We have no way to evaluate the likelihood that some party knows d for Juniper's non-standard Q point, except to note that Juniper does not appear to have used any of the recommendations presented in the NIST standard [82]. Based on the conclusions of Juniper's 2012 vulnerability report [80], however, it does seem reasonable to assume that the 2012 attacker-generated Q' was maliciously generated.

4.9.1 Questions and answers

In Section 4.3, we asked four questions which we reproduce here, along with their answers:

1. Why does a change in Q result in a passive VPN decryption vulnerability? *If an attacker is able to replace Q with a value of their choice, they can predict the (EC)DH keys which are used to establish the VPN traffic keys, thus decrypting VPN traffic.*
2. Why doesn't Juniper's use of X9.31 protect their system against compromise of Q? *In the default configuration, the X9.31 PRG is never run, and Dual EC output is sent directly to the network in the form of nonces. These nonces can be used to extract the Dual EC state and predict future outputs.*
3. What is the history of the PRG code in ScreenOS? *Version 6.1 contained a conventional X9.31-based PRG which was replaced with a two-stage PRG based on Dual EC in version 6.2. Version 6.2 also contained a number of other changes to the PRG and IKE code which combine to make ScreenOS vulnerable to a Q-replacement attack.*
4. How was Juniper's Q value generated? *We are unable to answer this question. In normal cases, the distribution of a maliciously-generated Q is identical to the distribution of a random Q. There do, however, exist techniques for* verifiably *generating safe Q points such that no attacker is likely to know the discrete log (one such procedure is described in [95]). If Juniper employed such a technique, it is still*

[95] Accredited Standards Committee (ASC) X9, Financial Services. *ANSI X9.82-3-2007: Random Number Generation, Part 3: Deterministic Random Bit Generators*. 2007

possible that a thorough review of historical documentation might provide evidence that such a technique was used.

5. Is the version of ScreenOS with Juniper's authorized Q vulnerable to attack? *We are unable to answer this question definitively. As demonstrated in this paper, an attacker who knows the discrete log of Juniper's Q would be able to decrypt traffic. However, because we do not know the details of Juniper's Q-generation procedure, we are unable to determine if such attackers exist.*

4.9.2 Unanswered Questions

Much of the coverage of the Juniper disclosure has focused on the unauthorized changes made in 2012 to the randomness subsystem and in 2014 to the login code. By contrast, our forensic investigation of ScreenOS releases highlights the changes made in the 6.2 series, in 2008, as the most consequential.

These changes, which introduced Dual EC and changed other subsystems in such a way that an attacker who knew the discrete log of Q could exploit it, were, as far as we know, added by Juniper engineers, not by attackers. This raises a number of questions:

How was the new randomness subsystem for the ScreenOS 6.2 series developed? What requirements did it fulfill? How did Juniper settle on Dual EC What organizations did it consult? How was Juniper's point Q generated?

We are not able to answer these questions with access to firmware alone. Juniper's source code version-control system, their bug-tracking system, their internal e-mail archives, and the recollections of Juniper engineers may help answer them.

Despite numerous opportunities, including public questions put to their Chief Security Officer and a congressional hearing on this incident,[xxv] Juniper has either failed or explicitly refused to provide any further details.

[xxv] Online: https://oversight.house.gov/hearing/federal-cybersecurity-detection-response-and-mitigation/.

4.10 Related Work on Dual EC

The history of the Dual EC random number generator was described by Checkoway et al. [21]. By 2006, it was already clear that the generator output has biases in its output that make it unsuitable for deployment, through work by Gjøsteen [78] and by Schoenmakers and Sidorenko [77]. Shumow and Ferguson's presentation at the Crypto 2007 rump session [19] further made clear that someone in possession of the discrete logarithm of Q to base P and who saw raw output from the generator would be able to reconstruct its internal state and predict all future outputs. Nevertheless, Dual EC was adopted as part of NIST SP 800-90A standard [5], and was not withdrawn until 2015 [96], following reporting based on the Snowden documents [97] that suggested that the Dual EC backdoor might be intentional. A presentation by John Kelsey gives a postmortem of Dual EC standardization from NIST perspective [81].

Our analysis in Section 4.4 shows that Juniper adopted Dual EC in 2008. In 2013, NIST reopening SP 800-90A for comments led Juniper to publish a knowledge base article explaining that ScreenOS uses Dual EC, but "in a way that should not be vulnerable to the possible issue that has been brought to light," because of the custom Q and because Dual EC output is filtered through X9.31 [83]. As our analysis shows, at the time that Juniper made this statement, the Q value shipping in ScreenOS was *already* one introduced in the unauthorized 2012 change. In January 2016, Juniper announced that it would remove Dual EC from its ScreenOS products in "the first half of 2016" [98].

[96] Elaine Barker and John Kelsey. *NIST Special Publication 800-90A Revision 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Tech. rep. National Institute of Standards and Technology, June 2015

[97] S. Shane, J. Larson, and N. Pearlroth. *N.S.A. Able to Foil Basic Safeguards of Privacy on Web - The New York Times*. 2013

[98] Bob Worrall. *Advancing the Security of Juniper Products*. Online: http://forums.juniper.net/t5/Security-Incident-Response/Advancing-the-Security-of-Juniper-Products/ba-p/286383. Jan. 2016

CHAPTER 5

SILENCE ON SIDE-CHANNELS

This chapter presents a novel attack on the CTR_DRBG pseudorandom number generator, showing that side-channel security is as important for pseudorandom number generators as it is for other cryptographic primitives.

> "My words, like silent raindrops fell, And
> echoed in the wells of silence."
>
> *Paul Simon*



Contributions

This chapter draws on forthcoming work to be published in IEEE S&P 2020 under the title *"Pseudorandom Black Swans: Cache Attacks on CTR_DRBG"* coauthored with Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, Yuval Yarom . I conceived and executed the side-channel attack on CTR_DRBG with the work on SGX performed by Andrew Kwong and Eyal Ronen.

- I present the first security analysis of CTR_DRBG in the presence of side-channel leakage, showing that the PRG state of many popular implementations can be recovered via cache attacks (Section 5.5).
- I show that PRG reseeding algorithms in popular implementations are sometimes insecure. Combined with the above state recovery attack, I empirically demonstrate an end-to-end attack on TLS that recovers long-term client authentication keys if the TLS client connects to an attacker-controlled TLS server (Section 5.7).
- I present a novel differential cryptanalysis technique that exploits side-channel leakage from CTR_DRBG running inside an SGX enclave to recover the PRG state within three AES encryption operations (Section 5.8.2).
- I demonstrate an end-to-end attack on a TLS client within an enclave. The attack is capable of passively decrypting the TLS connections regardless of PRG reseeding (Section 5.8.6).
- Finally, I evaluate CTR_DRBG's popularity by scraping National Institute of Standards and Technology (NIST) Cryptographic Module Validation Program database. I show that CTR_DRBG was the most popular design at time of writing, supported by 68% of the implementations (Section 5.9).

5.1 Introduction

One of the SP 800-90A standardized pseudorandom number generators (PRGs) is CTR_DRBG. The design of the PRG appears to have been inspired by the earlier X9.31 block cipher design with modifications that resolve the earlier issue explored in Chapter 3. The design again revolves around a block cipher, however unlike the X9.31 design, the key is rotated after each request to the PRG.

In concurrent work, Woodage and Shumow [18] examine the PRGs standardized in NIST SP 800-90A [5], proving that these PRGs meet the definition of Forward-Security proposed by Bellare and Yee [99]. They then highlight that the Bellare model and subsequent models [9,12,22,100,101] fail to capture the unconventional design of the SP 800-90A PRGs. Specifically, they note that the models do not account for the fact that the standard permits the generation of multiple output blocks per generate call, and that state compromises within such a call provide introduce overlooked attack vectors. This leads to theoretical attacks on two standardized PRGs: Hash_DRBG and CTR_DRBG. As highlighted in Chapter 3, the output from vulnerable PRGs is often statistically indistinguishable from output from safe implementations, so long as the attacker does not possess the keying material. This makes detecting vulnerable designs more challenging.

As specified in NIST SP 800-90, the key to the block cipher used by the PRG is rotated only after each call to the output generation function, and not each call to the underlying AES primitive. As a result, the PRG may make multiple calls to AES with single key. The maximum number of encryptions that can occur without key rotation may be limited by an implementation, and is bounded above by the specification that allows a maximum output of 2^{19} bits in a single call to the PRG, corresponding to 4096 bytes or 2^{12} calls. The standard's justification for the limit is that "[w]ithout these limits, it becomes possible, in principle, for an attacker to observe enough outputs from CTR_DRBG to distinguish its outputs from ideal random bits." The standard makes no mention of the flaw described above.

The primary risk of this flaw is that an attacker who observes a sufficient number of decryption, and is able to monitor system parameters (cache access patterns, timings, power) may be able to extract the key,

[99] Mihir Bellare and Bennet Yee. *CT-RSA*, 2003.
[100] Boaz Barak and Shai Halevi. *ACM CCS*, 2005.
[101] Mario Cornejo and Sylvain Ruhault. *ACM CCS*, 2014.

before rotation occurs. As key rotation is deterministic and invertible, an attacker who compromises the state at generate call g_i is able to compute the key for calls g_{i+1} and g_{i-1}, and may be able to recover keying material generated using the outputs.

Since their introduction in the seminal works of Bernstein [102], Osvik, Shamir, and Tromer [103] and Percival [104], microarchitectural attacks that exploit contention on internal components to leak information have been used to violate nearly every security guarantee offered by computer systems. Indeed, in recent years there have numerous examples of side-channel attacks with diverse targets and vectors. These range from attacks that extract cryptographic keys from keystroke timing [105,106] via CPU caches, attacks that exploit transient execution for breaking fundamental OS isolation guarantees [107–111], and even attacks that exploit limitations in memory hardware to change or read the contents of stored data [112–116]. Side-channel resistance is among the key security properties demanded of implementations.

Much less is known, however, about the security of PRGs in the presence of side-channel leakage. While backtracking resistance and prediction resistance are stated to be among the main security goals of the designs in NIST PRG recommendations (NIST SP 800-90A), the standard does not consider the impact of side-channel attacks on these goals. Although some initial evidence [117] already indicates the possibility of exploiting side-channel vulnerabilities in PRG seeding, a systematic exploration of side-channel leakage from PRG implementations has not been performed. Thus, this chapter explores the following main question:

[102] Daniel J Bernstein. *Cache-timing attacks on AES*. Online: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005
[103] Dag Arne Osvik, Adi Shamir, and Eran Tromer. *CT-RSA*, 2006.
[104] C Percival. *BSDCAN*, 2005.
[105] Yuval Yarom and Katrina Falkner. *USENIX Security*, 2014.
[106] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. *USENIX Security*, 2015.
[107] Paul Kocher et al. *IEEE S&P*, 2018.
[108] Jo Van Bulck et al. *USENIX Security*, 2017.
[109] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. Tech. rep. 2018
[110] Moritz Lipp et al. *USENIX Security*, 2018.
[111] Claudio Canella et al. *Proceedings of the 28th USENIX Security Symposium*, 2019.
[112] Yoongu Kim et al. *ISCA*, 2014.
[113] Naghmeh Karimi et al. *TACO*, 2015.
[114] Yu Cai et al. *HPCA*, 2017.
[115] Anil Kurmus et al. *USENIX WOOT*, 2017.
[116] Andrew Kwong et al. *IEEE S&P*, 2020.
[117] Yinqian Zhang et al. *ACM CCS*, 2014.

Are common PRG designs susceptible to microarchitectural side-channel attacks? What are the security implications of such leakage and how can the attacker exploit it?

Unfortunately, this chapter gives a positive answer to the above questions. CTR_DRBG is the most popular PRG design out of those recommended in NIST SP 800-90A, and is supported by 68% of validated implementations in NIST's Cryptographic Module Validation Program (CMVP). On the first question, this work shows that CTR_DRBG is vulnerable to state compromise attacks because some popular implementations still use a non-side-channel-resistant implementation of the underlying block cipher. On the second question, this work shows that several popular CTR_DRBG implementations fail to properly reseed the PRG in many situations, enabling feasible attacks against prediction resistance. Furthermore, this work demonstrates that use of Intel SGX permits a very strong *blind* state recovery attack in as few as three encryptions, without the attacker having access to PRG output. I demonstrate end-to-end attacks on the CTR_DRBG implementations used by OpenSSL's Federal Information Processing Standard (FIPS) module, NetBSD, and FortiOS, allowing an attacker targeting TLS connections to recover session secrets and long-term ECDSA keys used for client authentication, and under SGX, to passively decrypt connections.

The Use of T-Table AES. T-table AES is a performance-oriented AES implementation that uses table lookups to compute the state transitions between individual encryption rounds. Unfortunately, because these lookups are key-dependent, T-table AES has become the canonical example of cache side-channel leakage [103,118,119].

While the use of T-table AES for encryption and decryption operations has been greatly reduced in light of the threat posed by side-channels and the availability of AES-NI hardware, similar lessons do not seem to have been learned for the case of random number generation. Remarkably, even after more than a

[118] Joseph Bonneau. *Robust Final-Round Cache-Trace Attacks Against AES*. Cryptology ePrint Archive, Report 2006/374. 2006

[119] Michael Neve and Jean Pierre Seifert. *SAC*, 2007.

decade of attacks [102,103,120–122], I show that unprotected and leaky T-tables are still used for encrypting the counter inside CTR_DRBG by the following popular implementations:

- The OpenSSL 1.0.2 FIPS Module uses T-Table AES for CTR_DRBG. I note that use of this library is the *only* way to obtain U.S. government certification for a cryptographic module without submitting to the expensive and time-consuming validation process.
- The NetBSD kernel uses CTR_DRBG with a T-Table AES implementation as the system-wide random number generator.
- The FortiOSv5 network device operating system uses the same vulnerable CTR_DRBG implementation as NetBSD.
- mbedTLS-SGX, a port of the popular mbedTLS cryptography library to SGX [123].
- The nist_rng library [124], which is a library for random number generation used by open source projects such as libuntu (a C implementation of NTRUEncrypt), the XMHF hypervisor, as well as others.

CTR_DRBG State Recovery. By adapting previous work on AES encryption [119] to the PRG setting, this work extend the work of Woodage and Shumow [18] to show how an attacker who observes the cache access patterns of CTR_DRBG-based random number generation can recover the PRG's state using about 2000 bytes of the PRG's output. I then empirically demonstrate how a client that connects to a malicious TLS sever can be coerced to provide enough PRG output that an attacker can recover the PRG state used during the Transport Layer Security (TLS) handshake by concurrently observing the PRG's cache access patterns.

Extracting the Client's TLS Authentication Keys. Next, I show that NetBSD's kernel, OpenSSL's FIPS module and FortiOS fail to reseed the PRG with a sufficient amount of entropy. Thus, by using a moderate amount of brute forcing for the client entropy, the attacker can wind forward the client's PRG and recover the ECDSA nonce used by the client to authenticate herself to the malicious TLS server. Finally, using the recovered ECDSA nonce and the signature produced by the client during the TLS

[120] David Gullasch, Endre Bangerter, and Stephan Krenn. *IEEE S&P*, 2011.
[121] Samira Briongos et al. *Applied Sciences*, 2019.
[122] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. *ACM CCS*, 2012.
[123] Fan Zhang. *mbedtls-SGX*. Online: https://github.com/bl4ck5un/mbedtls-SGX. GitHub, 2018
[124] Henric Jungheim. *NIST SP 800-90Ar1 CTR_DRBG*. Online: https://henric.org/random/#nistrng. 2019

handshake, the attacker can recover the client's long term authentication keys. With authentication key in hand, the attacker can impersonate the client in future TLS connections.

State Recovery Without a Malicious TLS Server. The above attack on TLS requires the victim client to connect to a malicious TLS server, allowing the attacker to observe sufficient output generated by the client's CTR_DRBG implementation while simultaneously observing the client's cache access patterns across many AES encryption operations. Tackling this limitation, this work demonstrates a novel differential cryptanalysis attack exploiting side-channel leakage from T-table based CTR_DRBG running inside an SGX enclave. This attack leverages the fact that CTR_DRBG encrypts an incrementing counter. The technique is capable of extracting the PRG's state from only three AES encryption operations, without requiring the attacker to observe the PRG's output, thus eliminating the need for the TLS client to connect to an attacker-controlled server. Also note that this type of attack might also be applicable to other settings with similar constraints such as GCM-SIV [125].

Breaking TLS Connections With High-Entropy PRG Reseeding. Finally, note that any call to CTR_DRBG for random byte generation must use at least three AES encryption operations, producing the cache access information required by the differential cryptanalysis state-recovery technique. As this vector does not require the TLS client to connect to an attacker-controlled server, the attacker can recover the PRG state on *any* request for random bytes, regardless of how the implementation reseeds the PRG. This work demonstrates recovery of the premaster secret, master secret, and symmetric encryption keys for any TLS connection made by mbedTLS-SGX (a port of mbedTLS to SGX [123]) to any TLS server. In particular, this work demonstrates passive session decryption by observing cache access patterns made by mbedTLS-SGX.

5.2 Background

To begin, this work presents an overview of the AES symmetric block cipher, and an overview of the two cache side channel attacks used in this chapter.

[125] Shay Gueron and Yehuda Lindell. *ACM CCS*, 2015.

5.2.1 AES

AES encryptions and decryptions can be decomposed into four operations (AddRoundKey, SubBytes, ShiftRows, and MixColumns). Performance-optimized software implementations usually use a series of lookup tables known as "T-tables" to combine the latter three operations. AES encryptions and decryptions can be decomposed into rounds, which use round keys derived from the secret key to transform the input into a sequence of states. The state at each round is used to index into the T-tables, and the results are XORed with the round key to produce the state for the next round. The final round of AES uses a different T-table from earlier rounds as there is no MixColumns operation in that round. Unfortunately, by observing the memory access patterns to these tables, an attacker can recover the cipher's secret key within only a few encryptions. Indeed, starting from Osvik, Shamir, and Tromer [103], there has been a large body of work on attacking table-based AES implementations [106,120,126–128].

Most modern processors include CPU instructions that perform AES encryptions and decryptions in hardware. In addition to improving performance, these instructions do not rely on table lookups from system memory, thereby mitigating side-channel risks. Although hardware AES is widely implemented in modern desktop processors, many cryptographic libraries still use software-only implementations of AES in a variety of cases.

5.2.2 Cache Attacks

This work contributes to a long line of cache-based side-channel attacks. These attacks have yielded varied and robust mechanisms [129–131] for breaking cryptographic schemes using information leakage from cache timings. Popular targets have included digital signature schemes[132,133] and symmetric ciphers [103,105,134], despite the inclusion of countermeasures in popular cryptographic implementation libraries

[126] Gorka Irazoqui et al. *RAID*, 2014.
[127] Ning Zhang et al. *ePrint*, 2016.
[128] Raphael Spreitzer and Thomas Plos. *COSADE*, 2013.
[129] Daniel Gruss et al. *DIMVA*, 2016.
[130] Eran Tromer, Dag Arne Osvik, and Adi Shamir. *Journal of Cryptology*, 2010.
[131] Craig Disselkoen et al. *USENIX Security*, 2017.
[132] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. *USENIX Security*, 2014.
[133] Leon Groot Bruinderink et al. *CHES*, 2016.
[134] Rishabh Poddar, Amit Datta, and Chester Rebeiro. *Security Aspects in Information Technology*, 2011.

[135,136]. Recent literature has also begun to examine side-channel vulnerabilities in environments provided by trusted processor enclaves, particularly Intel SGX [108,137–141], which are designed to be more secure against even local attackers who are able to run unprivileged code.

Flush+Reload. Flush+Reload is a side-channel attack technique that consists of three steps. In the first step, the attacker *flushes* or evicts a memory location from the cache. The attacker then waits a while, allowing the victim to execute. Finally, in the third step, the attacker *reloads* the monitored memory location and measures the reload time. If the victim has accessed the memory location between the flush and the reload steps, the location will be cached, and the reload will be fast. Otherwise, the memory will not be cached and the reload will be slow. Flush+Reload has been used to attack symmetric [126] and public key [105,132,133,142,143] cryptography, as well as for non-cryptographic and speculative execution attacks [106–110,144–146].

Prime+Probe. While powerful, Flush+Reload relies on the victim and the attacker accessing the same memory location and is thus typically applied to OS-deduplicated pages in binaries and shared libraries. When shared memory is not available (e.g., for SGX), this work uses a different cache attack technique called Prime+Probe [103,130].

A Prime+Probe attack consists of three steps. In the first, the attacker primes the monitored cache lines by making enough memory accesses so that each way (group of cache lines fetched together) of the targeted cache sets is occupied by the attacker's memory value. In the second step, the attacker yields control to the victim process. In the final step, the attacker probes those same cache lines by reading from the corresponding memory locations and measuring their access times. If the victim accessed memory that

[135] The OpenSSL Project. *OpenSSL: The Open Source toolkit for SSL/TLS*. http://openssl.org/. 2003
[136] GnuPG Project. *GnuPG*. Online: www.gnupg.org. 2019
[137] Sangho Lee et al. *USENIX Security*, 2016.
[138] Ferdinand Brasser et al. *WOOT*, 2017.
[139] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. *SysTEX*, 2017.
[140] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. *IEEE S&P*, 2015.
[141] Yuan Xiao et al. *ACM CCS*, 2017.
[142] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. *ACM CCS*, 2017.
[143] Cesar Garcia and Billy Brumley. *USENIX Security*, 2017.
[144] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. *arXiv*, 2018.
[145] Stephan Van Schaik et al. *IEEE S&P*, 2019.
[146] Marina Minkin et al. *Fallout: Reading Kernel Writes From User Space*. 2019

mapped to the same cache lines, then the attacker will measure larger latencies for probes corresponding to those evicted cache lines.

5.3 CTR_DRBG

CTR_DRBG is a PRG design described in NIST SP 800-90A, depicted in Figure 8. It uses the encryption of an incrementing counter under a block cipher to generate outputs. The block cipher may be either 3DES with a 64-bit key or AES with a key of length 128, 192, or 256 bits. The design mixes in additional data at various stages. A derivation function (commonly the same block cipher under a different key) can optionally be used to extract entropy from the additional data. The implementations examined herein all used a derivation function.

Private State and Length Parameters. The private state $\mathcal{S}$ of the PRG is composed of the following:

- A key $K \in \{0,1\}^{keylen}$, with bit length $keylen$ matching that of the underlying cipher.
- A counter $V \in \{0,1\}^{\leq blocklen}$ that is incremented after each call to the block cipher, where $blocklen$ is the output length of the underlying block cipher.
- A reseed counter c that indicates when a reseed is required.

The PRG's nonce space $\mathcal{N}$ is $\{0,1\}^{seedlen}$ and entropy space is $\{0,1\}^{seedlen}$ where $seedlen = keylen + blocklen$.

PRG Instantiation. CTR_DRBG's instantiate function takes as input an entropy sample I and an arbitrary nonce N chosen by the implementation, of equal length. It computes a temporary value t as the output for the derivation function applied to I and N. It then calls a subroutine update, outlined in Algorithm 1, with inputs $K = V = 0$ and t as the additional input. The initial state $S_0 = (K, V, c)$ consists of the outputs (K, V) from update, and reseed counter $c = 1$.

State Update. Each of CTR_DRBG's functions call a subroutine update, outlined in Algorithm 1, that updates the internal state. The routine's input is a key K, counter V, and additional data $addin$. In Lines 4–6 the function increments the counter V and appends the encryption of V under key K to a

Algorithm 1 Update. The update routine is called by the other routines and passes the current state (and potentially additional input) into the underlying block cipher. It outputs new state $S = (K, V)$ composed of key K and counter V.

1: **function** UPDATE($K, V, addin$)
2: $temp \leftarrow null$
3: **while** **len**($temp$) $< seedlen$ **do**
4: $V \leftarrow (V + 1) \mod 2^{blocklen}$
5: $output_block \leftarrow$ encrypt(K, V)
6: $temp \leftarrow temp \| output_block$
7: $temp \leftarrow temp \oplus addin$
8: $K' \leftarrow$ leftmost$(temp, keylen)$
9: $V' \leftarrow$ rightmost$(temp, blocklen)$
10: **return** K', V'

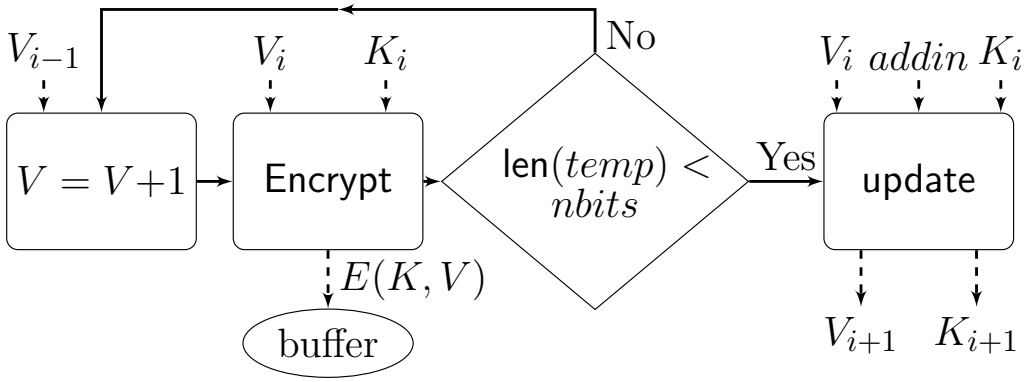


Figure 8: CTR_DRBG Output Loop. The central loop of the generate function increments the counter V, encrypts V under K, and adds the output to a buffer $temp$, repeating until $nbits$ have been generated. The function then updates the key and state before returning the contents of the buffer.

buffer $temp$. This process is repeated until $temp$ contains $seedlen$ bytes. The resulting buffer is then XORed with $addin$ (Line 7). Finally, in Lines 8–9 the function outputs the new key K' as the leftmost $keylen$ bits of the buffer, and new counter value V' as the rightmost $blocklen$ bits of the buffer, where $blocklen$ is the block length of the cipher.

Generating a Random Stream. A user generates output from the PRG by calling the generate function outlined in Algorithm 2. It takes as input the state S, the number of bits requested $nbits$, and a string $addin$, and outputs a string $nbits$ in length and an updated state S'. According to SP 800-90A, the $addin$ parameter "may be a means of providing more entropy for the DRBG internal state". This additional input is allowed to be public or private and may contain secrets if private. The specification

notes that "if the additional input is kept secret and has sufficient entropy, the input can provide more assurance when recovering from the compromise of the entropy input, the seed or one or more DRBG internal states". However, the specification does not include requirements for either secrecy or entropy for $addin$.

The generate function first checks if a reseed is needed, and if so, throws an error[xxvi] (Lines 3–4).

If the call included additional data $addin$, this data is first whitened by running it through the derivation function, and then it is used to update K and V through a call to update (Lines 5–7). Otherwise, $addin$ is set to a string of zeros (Line 9). On each iteration of the loop on Lines 11–14, the counter V is incremented. V is then encrypted under K and the result is appended to the output buffer. This process is repeated until enough output has been collected. On Line 16 the function calls update with $addin$ to update K and V again before the reseed counter c is incremented (Line 17). The function returns the new key, state, reseed counter, and output.

If the attacker compromises the key K and counter V between Lines 11–14 and is able to guess $addin$, she can predict the new key K' and counter V'. She can then predict future PRG outputs as well as future values of K and V. Note that the same symmetric key is used to generate all of the requested output, and the key is only changed at Line 16 after all blocks have been generated. This observation is a crucial element of the attack, since a long output buffer gives the attacker many opportunities to extract K via a side-channel. Indeed, SP 800-90A specifies that at most 65KB can be requested from the generator in a single call before a key change. This is presumably intended to limit a single state's exposure to an attacker. However, this work work demonstrates that state recovery attacks within this limit are still viable.

Reseeding. The reseed function is intended to ensure that high quality entropy is mixed into the state as required. The reseed function takes as input additional input $addin$, an entropy sample I, and a state S that consists of the key K, counter V, and reseed counter c. It calls the update subroutine

[xxvi] While the inclusion of an error message does not strictly adhere to my PRG definition, following Woodage and Shumow[18] this work assumes inputs are valid and omit consideration of errors from the analysis.

Algorithm 2 Generate. The generate function begins by throwing an error if the reseed counter exceeds the limit, and otherwise updates the state with the optional additional input, produces output by encrypting V under K, then increments V. The encryption and increment steps are repeated until the specified length of output has been produced. The state is then updated again, and the reseed counter is incremented.

1: **function** GENERATE($S, nbits, addin$)
2: parse (K, V, c) from S
3: **if** $c > reseed_interval$ **then**
4: **return** reseed_required
5: **if** $addin \neq Null$ **then**
6: $addin \leftarrow \mathsf{df}(addin)$
7: $(K, V) \leftarrow \mathsf{update}(K, V, addin)$
8: **else**
9: $addin \leftarrow 0^{seedlen}$
10: $temp \leftarrow Null$
11: **while len**$(temp) < nbits$ **do**
12: $V \leftarrow (V + 1) \mod 2^{blocklen}$
13: $output_block \leftarrow \mathsf{encrypt}(K, V)$
14: $temp \leftarrow temp \| output_block$
15: $out \leftarrow \mathsf{leftmost}(temp, nbits)$
16: $(K', V') \leftarrow \mathsf{update}(addin, K, V)$
17: $c' \leftarrow c + 1$
18: **return** $S = (K', V', c'), out$

on a derivation function taken over I and $addin$, which updates K and V. Finally, it resets the reseed counter c to 1 and returns the new key, counter, and reseed counter.

5.4 Cryptanalysis of CTR_DRBG

DRBG Security Proofs. Woodage and Shumow [18] note that historical analyses of the security claims in SP 800-90A [11,147–151] were limited by simplifying assumptions that were believed to be necessary due to nonstandard elements of the designs. Their analysis evaluated the standard's claims that the designs in the standard are both "backtracking resistant" and "prediction resistant". They provide robustness proofs that include backtracking and prediction resistance for both the HMAC and hash constructions, but were unable to do so for CTR_DRBG and instead identified an attack against the prediction resistance property.

Attacking CTR_DRBG. Bernstein [152] notes that to obtain prediction resistance after every random bit, the generate process must be called with only a single bit, incurring massive performance costs. Furthermore, SP 800-90A notes that "For large generate requests, CTR_DRBG produces outputs at the same speed as the underlying block cipher algorithm encrypts data". Woodage and Shumow [18] use this observation to propose an attack scenario where large amounts of CTR_DRBG output is buffered, setting the stage for a side-channel attack on the block cipher key. They give the following procedure for recovering output at $t+1$ from output r_t and key K_t that was compromised at time t:

1. **Counter Recovery From Output.**

   Attacker computes the state prior to the last update as $V'_t = \mathsf{decrypt}(K_t, r_t)$

2. **Generating S_{t+1}.**

   The attacker winds the generator forward by to retrieve the updated state by computing $K_{t+1}, V_{t+1} = \mathsf{update}(K_t, V'_t, addin_t)$

3. **Generating PRG Output r_{t+1}.**

   This state is now used to compute $r_{t+1} = \mathsf{generate}(K_{t+1}, V_{t+1}, addin_{t+1})$

[147] Matthew J Campagna. *Security Bounds for the NIST Codebook-based Deterministic Random Bit Generator.* 2006
[148] Wilson Kan. *Analysis of Underlying Assumptions in NIST DRBGs.* 2007
[149] Shoichi Hirose. *WISA*, 2008.
[150] Thomas Shrimpton and R. Seth Terashima. *ASIACRYPT*, 2016.
[151] Katherine Q. Ye et al. *ACM CCS*, 2017.
[152] Daniel J Bernstein. *Fast-key-erasure random-number generators.* Online: https://blog.cr.yp.to/20170723-random.html. July 2017

Overall Attack Complexity. Assuming that the attacker has access to K_t, the complexity of this attack depends only on the difficulty of the attacker guessing $addin_t$ and $addin_{t+1}$. While a naïve attacker might attempt to enumerate the entire space of $2^{seedlen}$ possibilities, I show that in practice implementations use low-entropy or predictable data such as timestamps for this parameter. I observed implementations that required as little as 2^{21} work to find the correct values for both $addin$ values.

I next evaluate the practicality of this attack in the context of cache side-channel attacks on popular CTR_DRBG implementations and evaluate the impact of these attacks on the security of TLS.

5.5 State Recovery Attack

I show that the attack described by Woodage and Shumow [18] is practical by recovering the CTR_DRBG state variables K and V via a cache side-channel attack against the underlying AES implementation. I begin with an overview of the popular implementations analyzed in this section.

5.5.1 Implementation Deep Dives

I examined the CTR_DRBG parameter choices of four implementations representing diverse use cases: the NetBSD operating system, the Fortinet FortiVM virtualized network device, and two versions of the OpenSSL cryptographic library. I identify limitations (if any) on the number of bytes that may be requested in a single call to the PRG, and highlight implementations' use of additional entropy. These parameters determine the viability of the state recovery attack. I also determine how frequently mandatory reseeds occur. If such a reseed incorporates sufficient entropy, an implementation may be able to 'recover' from compromise following the reseed.

FortiOS.

I analyzed FortiOS version 5, the second-most recent major release of Fortinet's network operating system for their hardware and virtual appliances. The operating system is an embedded Linux distribution with proprietary kernel modules that perform device-specific functionality. The software is used both on embedded devices and to operate VMs that perform virtualized network functions.

After reverse-engineering the operating system binaries, I discovered that FortiOSv5 replaces Linux's default implementation of `/dev/urandom` with the nist_rng library [124]. I note that FortiOSv5 behaved similarly to FortiOSv4 as described in Chapter 3, replacing the system's default PRG with a FIPS certified design. Both FortiOS v4 and v5 use OpenSSL to provide basic cryptographic functionality, which in turn relies on `/dev/urandom`. While the original OpenSSL will use an AES hardware implementation if it is available, Fortinet's override makes OpenSSL fall back to an unprotected T-table-based AES implementation based on the nist_rng library.

Finally, the FortiOS CTR_DRBG implementation does not use additional entropy on each update and has no explicit reseeding. It returns an error code if more than 99,999 blocks are cumulatively requested from the instantiated Deterministic Random Bit Generator (DRBG) over the course of its lifetime. It therefore lacks meaningful protection against state compromise.

NetBSD. The NetBSD operating system uses CTR_DRBG as the default source of system randomness. The kernel uses the nist_rng library with 128-bit AES as the default underlying cipher. I examined the kernel source code and single-stepped through a running kernel to verify my findings. As in the FortiOS case, the AES implementation is software-based with unprotected T-Table accesses, based on the nist_rng library. However, the OS limits requests to a maximum of 512-bytes from the PRG in a single call, increasing the difficulty of my proposed attack.

On each `generate` call, the state is updated using additional entropy from `rdtsc`, a high resolution CPU counter. NetBSD uses a 32-bit time-counter as additional entropy for each call to `generate`. NetBSD schedules an additional reseed after $2^{31} - 2$ calls to the PRG.

OpenSSL FIPS Module. I examined the OpenSSL FIPS module, which supports only OpenSSL 1.0.2. This implementation is one of a small number of libraries that a manufacturer can use to be FIPS compliant without submitting the entire product for certification [153]. The module uses CTR_DRBG with a user configurable key length. Notably, while OpenSSL 1.0.2 FIPS uses hardware instructions for AES encryption, the CTR_DRBG implementation uses a lower-level interface for AES. Instead of

[153] Matthew D Green. *Twitter Thread on OpenSSL*. Online: https:/twitter.com/matthew_d_green/status/1115013260783255558

selecting the best implementation available (as the AES interface used for encryption does), the lower-level interface used by CTR_DRBG uses a hand-coded T-Table AES implementation. On each generate call, the state is updated using the time in microseconds, a counter, and the PID. The FIPS module reseeds the PRG after 2^{24} calls to generate.

OpenSSL 1.1.1. The default PRG in OpenSSL 1.1.1, the most recent major release as of this writing, is a CTR_DRBG implementation forked from the OpenSSL FIPS code base. It defaults to 256-bit AES with user-configurable support for 128-bit and 192-bit AES. Unlike version 1.0.2 it *does* default to using hardware instructions for AES, so it is not vulnerable to my side-channel attack.

5.5.2 Side-Channel Attacks on AES-128

T-Table AES is the canonical target for cache side-channel attacks. Starting from Bernstein [102] many works [103,106,120,126,127] have demonstrated key extraction from cache access patterns of table-based implementations.

Since CTR_DRBG uses T-Table AES as its underlying cryptographic primitive, I implemented the attack of Neve and Seifert [119] on the last encryption round of AES in order to extract the AES key from the CTR_DRBG's cache access pattern.

5.6 Cache Attack Details

In this section, I present the details of the state recovery attack. In the synchronous model of Osvik, Shamir, and Tromer [103], an attacker observes the plaintext and is able to probe the cache state immediately before triggering an encryption with an unknown key. The attacker is also able to probe the cache state immediately after each encryption. Observing the cache access patterns caused by the first round of AES during a few encryption operations is sufficient to recover the key [103].

Attacking the Last Round of AES. Adopting the synchronous model of past work [103,118,119] I target the final round of AES, with attacker-observed ciphertext, rather than plaintext.

Implementations commonly use a different T-Table for the final round of encryption, allowing us to measure last round table accesses independently of earlier round accesses. Let q_i be the ith byte within the T-table, c_i be the ith ciphertext byte, and let k_i be the ith byte of the last round key. From the definition of T-table AES one knows that $c_i = T[q_i] \oplus k_i$ where T is the final round table. Thus, an attacker who observes c_i and determines q_i by monitoring the cache for accesses can solve this equation for the key byte, yielding $k_i = c_i \oplus T[q_i]$.

Handling Missing Information. While the attack outlined above works when the attacker has perfect visibility over q_i and i, on a real system the attacker does not directly observe q_i. Instead, she identifies a contiguous set of bytes that are fetched into the cache together (a cache line, typically 64 bytes) and thus loses information about some of the least significant bits of q_i. On my test machine, each access corresponded to sixteen different possible values for q_i, as each final T-Table byte is stored four times, in a 4-byte integer, sixteen of which are in each cache line. Further, the attacker does not know i, as she does not know which cache access produced which ciphertext byte. Thus, in order to obtain a candidate key byte k_i, the attacker must somehow *guess* the value of q_i from the table indexes accessed in the last round as well as guess the missing 4 bits from q_i. As one expects about 11 distinct indexes to be accessed in the last round [119], this results in about $11 \cdot 2^4 = 176$ candidate values for each k_i, out of 256 possible candidates.

Notice however, that across many independent encryptions of different plaintexts under the same key, the correct value for every k_i, $i = 0, \cdots, 16$ should *always* appear in the list of candidates. In contrast, one expects incorrect candidates to be uniformly distributed. Thus, if an attacker sees a large number of encryptions, she can combine the information obtained from them to retrieve the AES key. Let

$$\mathsf{hit}(q, j) = \begin{cases} 1 & \text{if } q\text{-th cache line accessed in } j\text{-th trace} \\ 0 & \text{otherwise} \end{cases} .$$

Following Neve and Seifert [119], the attacker counts cache hits that could correspond to each possible key byte value k from `0x00` to `0xFF` for each position i and stores the count in a table $\mathcal{S}$:

$$\mathcal{S}\left[i\right]\left[k\right] = \sum_{j=0}^{n} \sum_{q=0}^{\ell} \sum_{\substack{b=0 \\ T[2^m \cdot q+b] \oplus c_i = k}}^{m} \mathsf{hit}(q, j) \tag{5.1}$$

with ℓ the number of cache lines, m the number of bytes per cache line, and n the number of traces. As analyzed by Neve and Seifert [119] and Bonneau [118], the i-th byte of the last round key is then the value of k such that $\mathcal{S}\left[i\right]\left[k\right]$ is maximal.

5.6.1 Obtaining Trace Data

I describe how I mount Flush+Reload against CTR_DRBG. Begin by recalling that the attack of Neve and Seifert [119] outlined in Section 5.5.2 requires the attacker to gather ciphertexts paired with the corresponding traces of the cache state following the encryption operation that produced that ciphertext.

Matching PRG Output. To recover the AES key, an attacker must match each ciphertext to a trace taken in the interval following the encryption that produced it, but before the subsequent encryption. In the synchronous model of Osvik, Shamir, and Tromer [103] where the attacker triggers encryption operations directly, this matching is trivial. However, in my setting, a request for random bytes initiates a rapid series of encryptions. If the attacker's probes take a long time compared to an encryption operation, the attacker cannot easily interleave probes. This difficulty is exacerbated by the fact that encryptions vary in duration due to other system activity, making the naive strategy of probing at evenly spaced intervals fail to produce matching traces and ciphertext pairs.

Tickers. In order to use the synchronous setting analysis of Osvik, Shamir, and Tromer [103], I align traces and ciphertexts by using what I term "tickers". Tickers are frequent cache probes that measure how long it takes to access cache lines that contain program instructions. A cache hit on a ticker gives the attacker a signal she can use to determine whether to probe the cache lines containing the T-Table used in the last AES encryption round. In my case, I set two tickers. The first ticker queries instructions at the start of the encryption code (as loaded into the process' address space), and the second queries

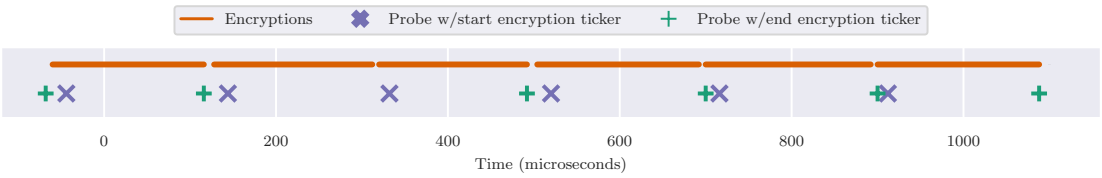


Figure 9: Probe Alignment with Encryptions. Probes do not perfectly align with the start and end of encryptions. Ideally, the start and end of an encryption probe should follow shortly after the start and end of an encryption. However, fluctuations in encryption duration and ticker timing accuracy cause misalignments. The problem is illustrated at $\approx 380\mu$s, where no end encryption ticker is visible, and at $\approx 900\mu$s where the end encryption ticker appears past the start of the next iteration.

instructions at the end of the encryption code. When either ticker is triggered, I probe the T-Table cache lines, ideally measuring cache state before and after encryptions.

Handling Drift. While tickers provide some signal, as depicted in Figure 9, variations in how the probe process is scheduled with respect to the victim process introduce imperfections in the signal provided by the tickers. Therefore, I also use timing heuristics to match traces to corresponding ciphertexts. More specifically, I iterate through the traces collected, and keep a counter identifying the next ciphertext to be matched to a trace. Then, for each trace, I either match it to the current ciphertext and increment the counter or discard it. I base this decision on the accompanying ticker and timing data.

The default case is to match the trace and ciphertext only if the ticker indicating a recent end-of-encryption event was triggered for that trace. However, to account for false negatives, the ticker indicating a recent start-of-encryption event is used if the interval between the last matched trace's timestamp and the current trace's timestamp exceeds a threshold I determined empirically. Similarly, if neither ticker was triggered, but the elapsed time is greater than another empirically-determined threshold, I match the trace and ciphertext.

Finally, using a ticker to determine when to start collecting traces may cause the attacker miss some traces belonging to the initial encryptions. I overcome this by running the key recovery algorithm with each possible set of matchings, for a small number of potential initial matches.

Overcoming Prefetching. Modern CPUs attempt to learn a program's cache access pattern and fetch data into caches before this data is actually needed. This data prefetching frustrates cache side-channel

attacks against T-Table AES by reducing the extent to which a recorded cache hit corresponds to an actual—rather than predicted—access. If an entire AES T-Table is preemptively fetched into memory, a naïve cache side-channel attack will not succeed because the attacker will record cache hits for every memory line.

I mitigated the effect of the prefetcher by accessing cache lines in an irregular order, using the pointer chasing technique of Osvik, Shamir, and Tromer [103]. This reduces the ability of the prefetcher to predict cache accesses and therefore prefetch those lines.

Performance Degradation. If the time it takes to probe the cache state is too long relative to the duration of an encryption, an attacker will not be able to generate traces that accurately capture the state of the cache after each encryption. Allan et al. [154] showed that this difficulty could be mitigated by continuously flushing cache lines containing victim program instructions, so that the victim process was significantly slowed down. Flushing cache lines requires the victim to repeatedly fetch code from main memory, increasing access times. On my system, this slowed down the average duration of an encryption from 2μsec to 32μsec, giving us a large 34μsec window between successive last AES rounds for cache probing.

Validating Key Candidates. In my setting, plaintexts encrypted within a single call to the CTR_DRBG `generate` function are sequential integers, providing a simple test to determine the correctness of a recovered key. Given a series of ciphertexts and a candidate key, I validate the key by decrypting the PRG output and checking if the plaintexts form a successive series of integers. The final integer in the sequence is the last counter value before the state is updated at the end of the procedure. Given the recovered key K, counter value V, and a valid guess for $addin$ (if any is used), the subsequent state and output of CTR_DRBG can be computed by executing the `update` subroutine.

5.6.2 Evaluation of State Recovery

Attack Scenario. I consider an attacker who can execute unprivileged code on a target machine. The attacker springs into actions when a victim process on the same machine uses CTR_DRBG and makes a

[154] Thomas Allan et al. *ACSAC*, 2016.

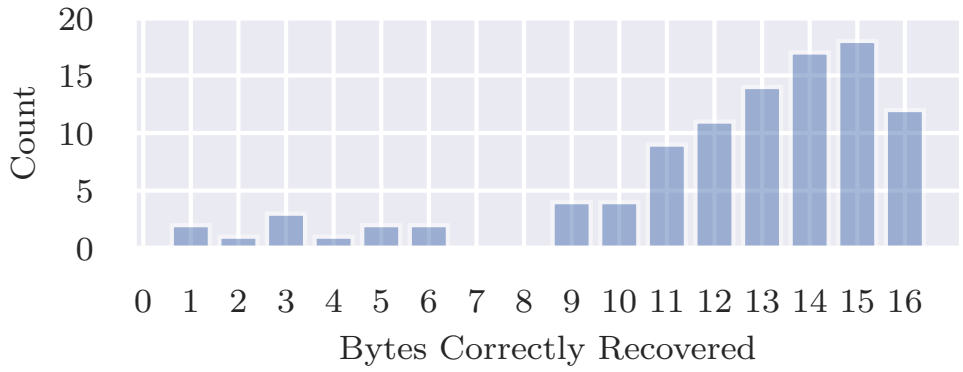


Figure 10: Key Recovery Rate with Prefetcher Enabled. With the prefetcher enabled, the state recovery technique often recovers only a subset of the full 16-byte AES key. I here depict the frequency with which a given number of bytes were recovered, across 100 trials.

call to generate, requesting about 2 KB of pseudorandom output. The attacker then uses Flush+Reload to monitor cache accesses during the AES operations inside the CTR_DRBG, and recovers the PRG state using the techniques described above. The experimental setup instantiates this scenario in a concrete setting.

Targeted Software. I targeted OpenSSL 1.0.2 configured to use the nist_rng library with AES128 as the underlying block cipher for the PRG. Beyond the implementations mentioned in Section 5.5.1, the nist_rng library is used by libuntu (a C implementation of NTRUEncrypt), the XMHF hypervisor, among others. As mentioned before, the nist_rng library uses a leaky T-table based AES implementation and does not support `AES-NI` hardware instructions.

Hardware. I performed the experiments on a desktop equipped with an Intel i7-3770 Quad Core CPU, with 8GB of RAM and 8MB last level cache. The machine was running Ubuntu 17.10 (Linux Kernel 4.13.0). To ensure fair comparison, I fixed the initial state of the random number generator to be the same uniformly sampled state for all the experiments described in this subsection.

Empirical Results. In 100 trials with the prefetcher disabled I was always able to recover the state, with an average false positive rate of 4.58% and false negative rate of 5.01%. As in Figure 10, with the

prefetcher enabled the attack succeeded in 12.0% of trials with average false positive rate 28.5% and false negative rate 1.94%. State recovery took an average of 19s in both cases, with hardware as above.

5.7 Attacking TLS

In this section, I put the side-channel attack in context and show how recovering the PRG state from CTR_DRBG leads to the attacker being able to compromise long-term TLS authentication keys. I begin with necessary background on TLS and cryptographic primitives.

5.7.1 RSA Background

RSA is a public-key encryption method that can be used as a key exchange method in TLS 1.2 and earlier. RSA is not included as a key exchange mechanism in TLS 1.3.

RSA Cryptosystem. An RSA public key consists of a public encryption exponent e and an encryption modulus N. The private key is the decryption exponent d, which satisfies $d = e^{-1} \bmod \phi(N)$, where $\phi(N) = (p-1)(q-1)$ is the totient function for an RSA modulus $N = pq$.

RSA Padding. An RSA-encrypted key exchange message begins by padding the message using PKCS#1 v1.5 [155] padding as depicted in Figure 11. PKCS#1 v1.5 padding is not CCA-secure and has led to numerous cryptographic attacks against RSA in practice [156,157]. Yet, it remains by far the most common padding method where RSA encryption is still used, including versions of TLS prior to 1.3.

Let m be a message to be encrypted, and $\text{pad}(m)$ be the message with PKCS#1v1.5 padding applied. The encryption m is the value $c = (\text{pad}(m))^e \bmod N$. The padded message $\text{pad}(m)$ can be recovered by the decrypter by computing $\text{pad}(m) = c^d \bmod N$. In normal RSA usage, the decrypter then verifies that the padding is correctly formatted, and strips it off to recover the original message m.

[155] FX Breu, S Guggenbichler, and JC Wollmann. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313. Internet Engineering Task Force, 2008, 1–19

[156] Matthew D Green. *Wonk post: chosen ciphertext security in public-key encryption (Part 2)*. Online: https://https://blog.cryptographyengineering.com/2018/07/20/wonk-post-chosen-ciphertext-security-in-public-key-encryption-part-2/. 2018

[157] Daniel Bleichenbacher. *CRYPTO*, 1998.

00	02	Padding	00	48-byte PMS

Figure 11: PKCS#1v1.5 RSA Padding Structure. PKCS#1v1.5 RSA encryption padding appends a pseudorandom padding string to the message, together with some fixed bytes. The padding block is filled with $k - 3 - \ell$ non-zero bytes that are generated by a pseudorandom number generator, where k is the byte-length of the modulus and ℓ is the byte-length of the message to be encrypted.

RSA-PSS. RSA-PSS is a probabilistic signature scheme with a formal security proof[158]. The padding scheme is designed to avoid the flaws in PKCS#1 v1.5 padding. The scheme uses a sequence of hashing operations and mask generation functions to generate a padded message from a salt s and the input message m. The salt can in general be a maximum of $\mathsf{len}(m) + hLen$ bytes in length, where $hLen$ is the length of the hash function output. RFC8446 (August 2018) [159] updates TLS 1.2, adding optional support for RSA-PSS signatures [160], but specifies that "the length of the Salt MUST be equal to the length of the [digest] output".

5.7.2 ECDSA

ECDSA is a standardized public key signature algorithm [73]. The global parameters for an ECDSA key pair include a pre-specified elliptic curve C with base point G of order n. The signer's private key is a random integer $1 < d_A < n$ and the public key is $Q = d_A G$.

To sign a message m, the signer generates a random integer nonce $1 < k < n$. The signature is the pair $r = (kG)_{\mathrm{x}} \mod n$ and $s = k^{-1}(H(m) + rd_A) \mod n$, where x represents the x-coordinate of an elliptic curve point, and $H(m)$ is the hash of the message m using a collision-resistant hash function H. Next, if an attacker learns the value of the nonce k, she can compute the private key d_A from the signature as $d_A = (sk - H(m))r^{-1} \mod n$. I omit the details of the signature verification procedure, as they are orthogonal to my attacks.

[158] Bellare Mihir and Phillip Rogaway. *PSS: Provably Secure Encoding Method for Digital Signatures*. Submission to IEEE P1363. 1998

[159] E Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force, 2018

[160] B. Kaliski, J. Jonsson, and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Internet Engineering Task Force, 2016

5.7.3 TLS Handshake Protocol

I describe the TLS 1.0, 1.1, and 1.2 handshake protocols necessary for the attack. A TLS handshake begins with a ClientHello message containing a 32-byte nonce along with a list of supported cipher suites. The standard specifies that the nonce should consist of a four-byte timestamp and 28 bytes of raw output from a pseudorandom number generator. The ServerHello message contains a similar nonce and the server's choice of cipher suite. I specialize to the case of RSA key exchange with mutual authentication, an option that is enabled for higher-security deployments, for VPN-over-TLS, and other instances where the server needs assurance of the client's identity. For these cipher suites, the server then sends a Certificate message with its certificate chain, a CertificateRequest message, and a ServerHelloDone message. The client checks the server certificate, generates a 48-byte premaster secret (PMS) and encrypts it to the server's public key from the certificate. The PMS and padding formatting are shown above in Figure 11.

The client then sends the RSA-encrypted premaster secret in a ClientKeyExchange message, sends its own certificate in a Certificate message, and a CertificateVerify message containing a signature computed over a transcript of the handshake thus far, that proves it possesses the relevant private key.

Upon receiving the encrypted ClientKeyExchange, the server decrypts the message, verifies that the padding has the correct structure, and then extracts the premaster secret. After the server obtains the PMS, it verifies the client certificate. Both client and server then derive symmetric encryption and authentication keys by applying a key derivation function to the premaster secret and the client and server nonces. Both sides exchange messages to authenticate the handshake, then begin transmitting encrypted traffic.

5.7.4 Finding Randomness in TLS

The state recovery attack described in Section 5.5 required 1996 bytes of output from the random number generator. Thus, for the cache side-channel attack to work at the protocol level, I needed to find places in the handshake where a single random number generator call would request enough output for an attacker to feasibly carry out state recovery. I evaluated the TLS protocol for potential sources of large

or variable length randomness and settled on three possibilities: the ExtendedRandom TLS extension, RSASSA-PSS padding, and RSA PKCS#1 v1.5 padding.

ExtendedRandom TLS Extension. ExtendedRandom is a non-standard extension to TLS that was proposed to the IETF [161] to permit clients to request up to $2^{16} - 1$ bytes of randomness from the server. While my attacks (as well as those discussed in Chapters 3–4) may have been able to make use of the increased output from the server's generator to recover secret information, there are no known implementations with a functional implementation of this extension [161].

RSA-PSS. I evaluated whether the generation of the random salt for RSA-PSS signatures provided a viable attack vector. Under the PSS specification, for a message of 2^{14} bytes, the maximum salt length allowed is 2016 bytes, or 126 blocks of PRG output, sufficient for the state recovery attack. However, since RFC8446 [159] restricts the salt length when PSS is used in TLS1.2, an attacker in this context cannot observe enough encryptions from calls to the underlying PRG.

PKCS#1 v1.5 Padding in TLS. When a TLS handshake is performed with an RSA cipher suite, the client generates the 32-byte premaster secret and encrypts it to the server's RSA public key, transmitting it in the ClientKeyExchangeMessage. If the malicious sever uses a 16384-bit RSA modulus, the client must generate 2,013 padding bytes, equivalent to 126 blocks of PRG output. This is a sufficient number of blocks for us to mount the state recovery attack. I thus target this mode of TLS.

5.7.5 Targeting TLS Clients

Unlike the attacks of Checkoway et al. [21] and those described in Chapters 3–4, which compromise the server's PRG, I compromise the state of the PRG used by the TLS client, since the client is the party that generates the encrypted key exchange message. However, similar to those attacks, I use the recovered state to predict future outputs of the PRG. In my case, this allows us to recover the client's long-term authentication key.

[161] Matthew D Green. *The strange story of "Extended Random"*. Online: http://blog.cryptographyengineering.com/2017/12/19/the-strange-story-of-extended-random/. 2017

Attack Overview. I assume the client connects to a malicious attacker-controlled server supporting TLS 1.0, 1.1, or 1.2 that uses RSA for key exchange, and that the client uses ECDSA for digital signatures. I also assume that the attacker is capable of running unprivileged code on the client. Next, since the RSA PKCS padding generation procedure requires the client to generate pseudorandom bytes, the attacker can use the cache leakage traces collected during the generation of the PKCS padding to recover the client's PRG state via the method described in Section 5.5. With the client's PRG state successfully recovered, the attacker predicts the subsequent PRG output and thus is able to compute the ECDSA nonce that the client generates in the course of producing the digital signature for the CertificateVerify message. As outlined in Section 5.7.2, an attacker who knows the nonce used to generate an ECDSA signature can trivially recover the long-term private key used for client authentication, even if that key was generated in a secure manner. Recovering the signing key allows the attacker to impersonate the client. This may allow the attacker to access TLS-protected resources that are served only to an authenticated client. The attack proceeds as follows:

1. **Victim Client Connects to an Attacker-Controlled Server.** A client with an ECDSA certificate is manipulated into visiting a web page with an attacker controlled script. The script initiates TLS handshakes with RSA cipher suites, to an attacker-controlled server. The server transmits an RSA certificate and requests mutual authentication.
2. **Recovering PRG State.** The client's software encrypts the TLS premaster secret to the server's RSA public key, generating PKCS#1v1.5 padding proportional to the size of the certificate. The attacker simultaneously conducts the state recovery attack explored in Section 5.5.
3. **ECDSA Signature Generation.** The client transmits its certificate and generates a random nonce to sign the CertificateVerify message using ECDSA. The client then transmits the signed CertificateVerify message to the server.
4. **Recovering the Client's Nonce.** The attacker conducts an offline search for entropy and additional input parameters used by the PRG to generate the client's ECDSA nonce. The attacker checks the nonce candidates by recomputing the ECDSA signature and validating it against the signature transmitted by the client.

5. **Key Recovery.** Finally, once the attacker successfully recovers the nonce, she is able to compute the client's ECDSA private key and can impersonate the client.

Performing Nonce Recovery. In order to perform Step 4, the state of the client's PRG must be advanced to the point at which ECDSA nonce generation occurs. The attacker can only wind the generator forward, and at each call to the `generate` and `reseed` functions the attacker must guess the entropy and additional input parameters. Thus, the attacker must pay close attention to implementation-specific details surrounding the ordering of calls to the PRG.

I illustrate this challenge using OpenSSL 1.0.2 as a baseline implementation for the nonce-recovery attacks described in this and following sections. I then modify the OpenSSL codebase across the set of implementation choices made by the other evaluated implementations to assess the quality of said choices.

5.7.6 Using PKCS#1 v1.5 in OpenSSL 1.0.2 for Nonce Recovery

I begin by describing the steps performed by OpenSSL during the establishment of a TLS connection to generate the random PKCS#1 v1.5 padding and ECDSA nonce. For ease of reference, I label each step of these processes and then describe the end-to-end attack on OpenSSL 1.0.2.

1. **Initial Padding Generation.** The output of the PRG is fed into an n-byte buffer to be used for PKCS#1 v1.5 padding, where n is the length of padding required (in my case $n = 1996$). The state is updated twice, once before the bytes are generated and once after. State compromise occurs after the first call to `update`, but prior to the second.
2. **Padding Zero-Fill.** PKCS#1 v1.5 does not allow `0x00` bytes to be present in the random padding, so if there are z `0x00` bytes present in the PKCS output buffer, OpenSSL makes at minimum z more requests for output from the PRG, one for each byte. If any of these additional requests also result in a `0x00` byte, OpenSSL makes repeated requests to the PRG until the output is non-zero. The output from these requests is used to replace the null bytes in the padding to produce a valid non-null padding string under PKCS#1 v1.5. Within each request for random bytes, the PRG state is advanced twice. Both updates use the same underlying additional input.

3. **RAND_seed.** The ECDSA signing routine attempts to reseed the OpenSSL PRG via `RAND_seed`. The SHA256 hash of the TLS handshake transcript is used as external entropy.
4. **RAND_add.** A call to `RAND_add` is made as part of `bnrand`, which is used to generate a random integer in a given range. Time in seconds is used as external entropy.
5. **GenNonce.** OpenSSL then generates the ECDSA nonce.

   Within the call to the CTR_DRBG generate function, the state is updated before the nonce value is finally produced.

Notably, Steps 3 and 4 call functions from the OpenSSL's PRG API, which as discussed in Section 5.7.7 do not always perform the expected function of reseeding or updating CTR_DRBG.

Causing a Large Number of Random Byte Generations. To perform the attack, the attacker must observe side-channel leakage during the generation of a large amount of randomness. Moreover, to recover the PRG's state, the attacker must learn the values of the victim-generated randomness. In my attack scenario, the attacker could cause a victim client to connect to the attack server using a malicious script served by an ad network on a website the user would otherwise normally visit. The attacker's malicious server is configured to support only RSA key exchange, and deliberately serves a 16534-bit RSA certificate, which is the maximum size that OpenSSL will support without throwing an error during the handshake.[xxvii]

Next, while encrypting the premaster secret to the server's 16534-bit RSA public key to generate the ClientKeyExchange message for the TLS handshake, the client generates 1,996 bytes of PKCS#1v1.5 padding output, which, if using CTR_DRBG, gives the server an opportunity to conduct a side-channel attack against 125 AES encryptions. The attack server learns the value of the padding generated by the client by decrypting the padded RSA-encrypted message using its private key. The attacker then recovers the PRG state via the method described in Section 5.5, using the decrypted padding as the ciphertexts.

[xxvii] This is due to deliberate, hard-coded limits on message sizes that OpenSSL will accept, in the interest of preventing denial of service attacks [162,163].

[162] [OPENSSL-USERS] MAILING LIST. *Openssl software failure for RSA 16K modulus*. Online: https://mta.openssl.org/pipermail/openssl-users/2016-July/004056.html. 2016

[163] [OPENSSL-DEV] MAILING LIST. *[openssl.org #4063] Re: Client Hello longer than* 2^{14} *bytes are rejected*. Online: https://mta.openssl.org/pipermail/openssl-dev/2015-September/002860.html. 2015

The Problem of Padding Zero-Fill. As noted above, to comply with the PKCS standard, there must be no `0x00` bytes in this padding. OpenSSL complies by first generating padding of the total length required, and then replacing each null byte with output from further calls to the PRG, each used to byte. To encrypt to the malicious server's large certificate, OpenSSL generates 1,996 bytes of output for padding used as per Figure 11. In expectation, a ciphertext will have eight such `0x00` bytes that need to be replaced.

Next, for each `0x00` byte in the padding, the generator will have advanced an additional time. Since the attacker must brute force over the additional entropy added at each step, this increases the search space exponentially in the number of bytes generated in Step 2 to recover the final PRG state.

Bypassing RAND_add. However, as the initial 1996 padding bytes (generated during the initial Padding Generation step) have a uniform distribution over the 256 possible byte values, the probability of the padding not containing `0x00` is $(255/256)^{1996}$. I therefore expect that once every $2470 \approx 2^{11.3}$ TLS handshakes the padding generated after the Padding 1 step will not require additional calls to CTR_DRBG in the Padding 2 step to produce a valid PKCS#1 v1.5 padding string. Combining this with the success rate for state recovery in Section 5.6.2, an attacker can be expected to recover PRG state once in every 2^{18} handshakes.

Nonce Recovery. With the PRG state recovered, the attacker proceeds to recover the client's ECDSA nonce. Since the nonce is generated in a new call to the PRG, the PRG is reseeded between the adversary's state recovery attack and nonce generation. An attacker must therefore obtain the values used during RAND_seed and RAND_add (Steps 3 and 4). The exact strategy of recovering these values is implementation-specific.

5.7.7 Implementation Choices and Nonce Recovery

In this section I describe how implementations use the $addin$ parameter, and how they explicitly reseed the generator. I describe how this impacts my ability to recover the value of $addin$ and entropy used during RAND_seed and RAND_add (Steps 3 and 4) in Section 5.7.6.

FortiOS. FortiOS does not implement `RAND_seed` and `RAND_add`, and instead relies on the nist_drbg library's internal reseed counter. As a result, RAND_seed and RAND_add do not cause a state update, reducing the attack complexity.

Furthermore, as FortiOS does not not use the optional additional input for calls to generate, the PRG can be wound forward without the offline search for additional input.

Custom Parameters for FortiOS. I modify the FortiOS implementation to illustrate that even if it were to improve its reseeding and updating strategies, the implementation can be attacked in the absence of sufficiently high-quality entropy input. To evaluate this, I modified the FortiOS `RAND_METHOD` behavior to cause it to reseed during RAND_seed and RAND_add. Moreover, I added support for the additional data parameter, filling it with a microsecond timestamp to emulate OpenSSL FIPS.

OpenSSL FIPS. The OpenSSL 1.0.2 FIPS module also does not reseed the CTR_DRBG during RAND_seed and RAND_add. Instead, these calls add the entropy to a general entropy pool from which the PRG can later be reseeded with a call to reseed in compliance with SP 800-90A. I estimated the amount of entropy added during generate calls to be 12 bits.

OpenSSL 1.1.1. In OpenSSL 1.1.1 (the latest version at the time of writing) the maintainers rewrote much of the random number generation API. Due to the significant changes, this code was professionally audited twice[164,165], both times finding only minor flaws with the PRG implementation. The implementation gathers additional input from a variety of sources and feeds it into an entropy pool. These include system event timing data, time, thread ids and output from the OS or hardware random number generators. Given this complexity, I did not estimate the entropy added in reseeding.

The ECDSA nonce generation mechanism in OpenSSL 1.1.1 was also improved. The nonce is generated from a hash of the private key, along with the transcript, and PRG output. The inclusion of secret data

[164] Jean-Philippe Aumasson and Antony Vennard. *Audit of OpenSSL's randomness generation*. Online: https://ostif.org/wp-content/uploads/2018/09/opensslrng-audit-report.pdf. 2018

[165] Quarkslab SAS. *OpenSSL Security Assessment*. Online: https://ostif.org/wp-content/uploads/2019/01/18-04-720-REP_v1.2.pdf. 2019

ensures that even if the PRG is compromised, the nonce cannot be recovered. Together, these measures preclude both state and nonce recovery.

NetBSD. The NetBSD kernel provides a source of random numbers that can be used by a TLS implementation. I consider an implementation that, like FortiOS, chooses to source random numbers for OpenSSL from the system PRG without modification. NetBSD provides additional data in to CTR_DRBG in the form of the least significant 32 bits of the `rdtsc` cpu counter. If this counter is not available, NetBSD uses the kernel's current time in microseconds, and further falls back to an integer counter if the kernel clock is not yet running. It is not possible for applications to add further entropy as NetBSD does not externally expose the `reseed` and `update` functions, and thus I do not model any additional entropy introduced by `RAND_seed` and `RAND_add`.

5.7.8 Evaluation

In this section, I empirically evaluate the difficulty of extracting ECDSA signing keys from TLS clients given the different implementation choices described in Section 5.7.7. In order to evaluate the effects of different parameter choices on attack complexity, I reverse-engineered the FortiOS CTR_DRBG implementation and reimplemented it ourselves using the nist_rng library, so that I could easily adjust parameters and hook it into implementations. I modeled attack difficulty against the other implementations by adjusting *addin* and reseeding behaviors to match the descriptions in Section 5.7.7 of each implementation.

The Victim. For my victim TLS client, I used the sample TLS client code available in the OpenSSL documentation [166], configured to use mutual authentication and the nist_rng library with my choice of modeling parameters. I configured the client to authenticate using an ECDSA certificate with NIST P-256. For the ECDSA nonce, I used the raw PRG output, which matches the behavior of all implementations considered in Section 5.7.7, except OpenSSL 1.1.1.

The Malicious TLS Server. My malicious server was the default OpenSSL tool, instrumented to dump TLS transcripts and ECDSA signatures to the filesystem, and configured to support only RSA

[166] OpenSSL. *SSL/TLS Client*. 2018

Target	Sources	Search Space	Reduced Space	CPU Time
OpenSSL FIPS	time, PID,counter	2^{24}	2^{21}	30 minutes
NetBSD	rdtsc	2^{32}	2^{21}	30 minutes
FortiOS	none	0	0	N/A
Custom Params	time	2^{48}	2^{43}	200 years

Table 9: Nonce Recovery Search. I calculated the search space for the attack described in Section 5.7.7. I extrapolated custom parameter timing from smaller timings on the test machine. OpenSSL 1.1.1 is excluded from experimental evaluation due to its non-vulnerable nonce generation mechanism. The full search space corresponds to the search complexity of all possible timestamps of that size, and the reduced space corresponds to a search of one standard deviation from the mean required search, starting from the approximate timing of the encryption operation I gained from the timing attacks, calculated across 100 trials.

key exchange cipher suites with a 16384-bit RSA certificate, the largest allowed key size as discussed in Section 5.7.6.

PRG State Recovery for Winding Forward. After a TLS client connects to the malicious server, I use Flush+Reload to recover the PRG state, as described in Section 5.5. I then brute forced *addin* and additional entropy to recover the ECDSA nonce, which consists of raw PRG output.

My ability to wind the generator forward largely depends on the quantity of entropy injected between state recovery and nonce generation. Table 9 summarizes the entropy sources and brute force search space for each implementation.

Using Side-Channel Information for Space Reduction. I note that the attacker can use the same cache side-channel used for state recovery to reduce the search space over the additional entropy sources. By placing additional tickers and using timing data acquired during the state recovery process, I narrow down the set of timestamps or CPU counter values that I need to search. I empirically evaluate the amount of data that can gained through the instrumentation already in place for conducting state recovery in Table 9 as well. Note the entropy brute forcing is highly parallelizable, because after the SSL/TLS handshake has been performed, each element of the search space can be tested independently.

Empirical Results. The attack succeeded against FortiOS in negligible time (following state recovery) and against OpenSSL FIPS after thirty minutes (2^{21} work) using the hardware setup from Section 5.6.2.

For the custom parameters the search space was beyond my computational capabilities, and I terminated the search after approximately one hour of searching. I tabulate my results in Table 9. While tbe experimental results are limited by my CPU's speed of approximately 2^{22} elliptic curve scalar multiplications per hour, Vanhoef and Ronen [167] achieve a rate of 2^{35} operations per hour using a commodity GPU. I therefore anticipate that using their setup, the custom parameters search (2^{43} work) would be completed within two weeks.

Handling AES-256. To demonstrate key recovery under the constrained set of known ciphertexts available in the TLS setting of Section 5.7, I implemented the attack using AES-128. In Section 5.8, this work handles AES-256 in the SGX setting.

5.8 Attacking Full Entropy Implementations

The attack in Section 5.7 relies on both the ability to observe the output of the PRG and brute force the limited entropy of the state update. These are not fundamental requirements: by carrying out a higher-resolution cache attack, one can remove these limitations. This work demonstrates such a *blind* attack in which the attacker can observe the victim's cache access patterns but not the PRG output. Furthermore, the attack only requires observing two AES encryptions and thus is feasible even when the update entropy is too high to brute force.

However, to achieve this a stronger side-channel adversary is required: one who can observe the cache accesses during AES encryption with high temporal resolution. This model is plausible in the setting where the victim runs within an SGX enclave on a host with an attacker-controlled operating system. This scenario is within the threat model for SGX enclaves within which high resolution side-channel attacks are feasible[168–170].

This work begins with background on SGX, cache attacks on SGX, and the SGX threat model (Section 5.8.1). It then presents a novel differential cryptanalysis technique for exploiting side-channel

[167] Mathy Vanhoef and Eyal Ronen. *IEEE S&P*, 2020.
[168] Jo Van Bulck, Frank Piessens, and Raoul Strackx. *SysTEX*, 2017.
[169] Marcus Hähnel et al. *USENIX ATC*, 2017.
[170] Wenhao Wang et al. *ACM CCS*, 2017.

information (Section 5.8.2). Last, the work evaluates the attack against an SGX port of the mbedTLS library (Section 5.8.6).

5.8.1 Secure Enclave Technology

Intel Software Guard Extensions (SGX)[171] is an extension of the x86 instruction set that supports private regions of memory called *enclaves*. The contents of these enclaves cannot be read by any code running outside the enclave, including kernel and hypervisor code. SGX is designed to allow a user-level process to protect code and data from a highly privileged adversary, such as a malicious operating system or hypervisor.

Cache Attacks on AES Inside SGX. Although SGX protects the enclave from a malicious OS, it renders the code in the enclave *more* vulnerable to side-channel attacks. Specifically, the cache attack of Section 5.5 can only observe the overall access pattern over an entire encryption. However, when the victim runs in an SGX enclave, a malicious operating system can obtain much finer temporal resolution. This allows us to observe cache accesses after each of the 16 accesses to the AES T-tables in each of the encryption rounds [168,169].

Threat Model. Following previous work [108,138–140], in this section the work considers a root-privileged attacker who controls the entire OS. This is in agreement with the SGX threat model, where an enclave guarantees confidentiality and integrity, even against a malicious OS and hypervisor. Unlike the attack described in Section 5.7, this section does not assume that the TLS client in the enclave is willing to connect to a malicious attacker-controlled server, or uses imperfect PRG reseeding.

5.8.2 Differential Cryptanalysis via Side-channel Leakage

AES is a substitution-permutation cipher [172] that operates in a sequence of rounds on a 128-bit internal state S. Each round mixes the state and combines the mixed state with a round key. For a plaintext x, the initial state is $S_0 = x \oplus K_0$. Each consecutive round calculates $S_{j+1} = P(S_j) \oplus K_{j+1}$, where P is

[171] Matthew H. *Intel SGX for Dummies (Intel SGX) Design Objectives*. Online: `software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx`. 2013

[172] Alex Biryukov. In: *Substitution–Permutation (SP) Network*. Henk C. A. van Tilborg and Sushil Jajodia, eds. 1268–1268. Boston, MA: Springer US, 2011.

the state mixing function and K_j is the key for the j^{th} round. For efficient software implementation, the mixing step is implemented using four T-tables. Each byte of the state selects one entry from a T-table. As the T-table entries are 32 bits wide, a change in any given state bytes affects four consecutive bytes in the mixed state. For example, one can calculate the first four bytes of state S_{j+1} by:

$$S_{j+1,0..3} = T_0[S_{j,0}] \oplus T_1[S_{j,5}] \oplus T_2[S_{j,10}] \oplus T_3[S_{j,15}] \oplus K_{j+1,0..3} \tag{5.2}$$

As before, the cache attack targets accesses to these T-tables. Because one cannot distinguish between entries in the same cache line, the cache leaks only the four most significant bits of each byte of the state in each round. Let $\langle\rangle_U$ denote setting the four least significant bits of each byte to zero, then the leakage on byte k is $L_{j,k} = \langle S_{j,k}\rangle_U$. With a known plaintext x, one can use $L_{j,k}$ to recover the four most significant bits of every byte of K_0 because $\langle K_{0,k}\rangle_U = \langle x_k\rangle_U \oplus L_{0,k}$.

Unfortunately, in the blind attack one does not know x. Consequently, one cannot learn information on K_0 from the leakage of the first round. Instead, one can use the known difference between the plaintexts used in consecutive rounds of AES-CTR to recover the AES state. This is a direct artifact from the use of an incrementing counter as the plaintext. This is in close correspondence to the changes targeted in differential fault attacks [173]. This work combines a closely related analysis with side-channel leakage to form the basis of the attack. Once one has recovered the state, recovering the set of round keys, plaintexts, and ciphertexts is straightforward.

5.8.3 Attack Overview

Notation. This work uses the following notation:

1. $T_0..T_3$ is the array of 4 AES T-Tables, where $T_i[j]$ is the value in location j of Table i.
2. $\langle x\rangle_U$ denotes the value of x with the lower four bits (nibble) in each byte set to 0.

[173] Eli Biham and Adi Shamir. *CRYPTO*, 1997.

3. $L_{i,j,k}$ is the value leaked from the cache attack for byte k of round j in trace i. The leaked value is only the 4 most significant bits and the lower nibble is always 0.
4. $S_{i,j,k}$ is the real value of the state byte k of round j in trace i. $G_{i,j,k}$ is the current guess for this byte.
5. $R\Delta_{j,k}$ the value of the differential $S_{0,j,k} \oplus S_{1,j,k}$, and $\Delta_{j,k}$ is the current guess for this value.
6. $L\Delta_{j,k} = L_{0,j,k} \oplus L_{1,j,k}$ (lower nibble is always 0).
7. $K_{j,k}$ is the key value of byte k of round j.

Differential Analysis. By analyzing the difference between the state of two consecutive encryptions, one can recover state information that is independent of the round keys. In AES-CTR, for two consecutive plaintexts $x0$ and $x1$ one knows that $x1 = x0 + 1$, so with probability $(255/256)$ the two plaintexts only differ in the last byte by some value Δ_{ctr}. As the state of round 0 is simply the plaintext XOR with K_0, the plaintext difference is preserved and $R\Delta_{0,15} = \Delta_{ctr}$. Using Equation (5.2) one obtains:

$$
\begin{aligned}
S_{0,1,0..3} =& T_0[S_{0,0,0}] \oplus T_1[S_{0,0,5}] \oplus T_2[S_{0,0,10}] \\
& \oplus T_3[S_{0,0,15}] \oplus K_{i+1,0..3} \\
S_{1,1,0..3} =& T_0[S_{1,0,0}] \oplus T_1[S_{1,0,5}] \oplus T_2[S_{1,0,10}] \\
& \oplus T_3[S_{1,0,15}] \oplus K_{i+1,0..3} \\
=& T_0[S_{0,0,0}] \oplus T_1[S_{0,0,5}] \oplus T_2[S_{0,0,10}] \\
& \oplus T_3[S_{0,0,15} \oplus R\Delta_{0,15}] \oplus K_{i+1,0..3} \\
L\Delta_{1,0..3} =& L_{0,1,0..3} \oplus L_{1,1,0..3} = \langle S_{0,1,0..3} \oplus S_{1,1,0..3} \rangle_U \\
=& \langle T_3[S_{0,0,15}] \oplus T_3[S_{0,0,15} \oplus R\Delta_{0,15}] \rangle_U
\end{aligned}
\tag{5.3}
$$

As $\langle S_{0,0,15} \rangle_U = L_{0,0,15}$ one only needs to try the sixteen options for the lower four bits until finding a value that satisfies Equation (5.3) thereby recovering $S_{0,0,15}$ (See Algorithm 5). As $R\Delta_{0,15}$ is unknown, one is required to run Algorithm 5 with each possible value to retrieve the full set of candidates. However, as $R\Delta_{0,15} = x_0 \oplus x_0 + 1$ there are only eight possible candidates.

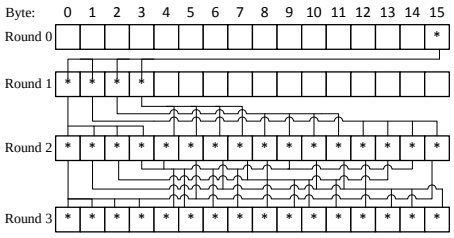


Figure 12: Single byte differential propagation in AES state.

Algorithm 3 Generating Candidates for Byte 0 of Round 1.

```
1: function ByteToRound1(L_{0,1,0}, Δ_{1,0}, LΔ_{2,0..3})
2:     GuessList1 ← Empty
3:     for Nibble ← 0 to 2^4 − 1 do
4:         G_{0,1,0} = L_{0,1,0} ⊕ Nibble
5:         Δ_{2,0..3} = T_0[G_{0,1,0}] ⊕ T_0[G_{0,1,0} ⊕ Δ_{1,0}]
6:         if ⟨Δ_{2,0..3}⟩_U = LΔ_{2,0..3} then
7:             GuessList1.append(G_{0,1,0}, Δ_{2,0..3})
8:     return GuessList1
```

Using Three or More Traces. The above attack requires only two traces to compromise the CTR_DRBG state. However, any request for PRG output causes at least three encryptions or four when AES-256 is used as the underlying block cipher.

The attack can be trivially extended to use the extra encryptions to more efficiently eliminate candidates, which aids in reducing the impact of noisy measurements.

5.8.4 Extended Differential Analysis

In a differential attack one can only recover state bytes that differ in the two encryptions. The attacks herein thus follow the "differential propagation" in the AES rounds as shown in Figure 12. This allows an attacker to recover one byte of state of round 0, 4 bytes of the state of round 1, and the state variables from round 2 and above.

From State to Key and Plaintext Recovery. Assuming one was able to recover complete state variables in rounds j and $j+1$, one can now recover the key for round $j+1$:

Algorithm 4 Generating Candidates for Bytes 0, 5, 10 and 15 of Round 2.

function BYTE0-5-10-15-ROUND2($L_{0,2,(0,5,10,15)}$, $\Delta_{2,(0,5,10,15)}$, $L\Delta_{3,0..3}$)
  GuessList2 ← Empty
  $IndxList$ ← [0,5,10,15]
  for $Guess$ ← 0 to $2^{16} - 1$ **do**
    $\Delta_{3,0..3}$ = 0
    for i ← 0 to 3 **do**
      Nibble = $(Guess >> (i * 4))\&0xf$
      $G_{0,2,IndxList[i]}$ = $L_{0,2,IndxList[i]} \oplus$ Nibble
      $\Delta_{3,0..3} \oplus= T_i[G_{0,2,IndxList[i]}]$
      $\Delta_{3,0..3} \oplus= T_i[G_{0,2,IndxList[i]} \oplus \Delta_{2,IndxList[i]}]$
      if $\langle \Delta_{3,0..3} \rangle_U$ = $L\Delta_{3,0..3}$ **then**
        GuessList2.append($G_{0,2,(0,5,10,15)}$, $\Delta_{3,0..3}$)
  return GuessList2

$$S_{i,j+1} = K_{j+1} \oplus P(S_{i,j}) \quad \text{and} \quad K_{j+1} = S_{i,j+1} \oplus P(S_{i,j})$$

As the AES key schedule for deriving the round keys is invertible, one can use any 128 bit round key to recover the original 128 bit AES key (one needs two consecutive round keys for 256 bit AES keys). From the recovered key and state one can calculate both the plaintext and ciphertext.

Iterative State Guess Elimination. In the beginning of step j of the attack there are one or more possible guesses for the values of the state bytes of round j. For each guess one enumerates all possible guesses for state bytes in round $j + 1$, eliminating guesses that do not satisfy the above differential propagation equations. The remaining guesses are used as input for the subsequent attack step. When one has a guess for the state of two full rounds, one can attempt plaintext recovery and verify that the plaintexts for a sequences of incrementing values.

Note that using 3 or more traces helps in eliminating wrong guesses, usually leaving just a single guess after each step.

The Full Attack.

Algorithm 5 Generating Candidates for the Last State 0 Byte.

```
1: function LastStateoByte(L_{0,0,15}, Δ_{0,15}, LΔ_{1,0..3})
2:     GuessList0 ← Empty
3:     for Nibble ← 0 to 2^4 − 1 do
4:         G_{0,0,15} = L_{0,0,15}⊕ Nibble
5:         Δ_{1,0..3} = T_3[G_{0,0,15}] ⊕ T_3[G_{0,0,15} ⊕ Δ_{0,15}]
6:         if ⟨Δ_{1,0..3}⟩_U = LΔ_{1,0..3} then
7:             GuessList0.append(G_{0,0,15}, Δ_{1,0..3})
8:     return GuessList0
```

Note that one can retrieve GuessList0 that contains all possible guess for $G_{0,0,15}$ and $\Delta_{1,0..3}$ using Algorithm 5. For each guess in GuessList0 the attacker then attempts to recover 4 bytes from the first round using a similar method. As each of the four bytes affect a different four bytes in the second round, one iterates through the same algorithm as in Algorithm 5 with each of these different values. In Algorithm 3 this work shows how to find the possible guesses for $G_{0,1,0}$. A similar function will find the possible guesses for $G_{0,1,1}$, $G_{0,1,2}$ and $G_{0,1,3}$. As the attacker may have more than one guess for each byte value, the full guess list GuessList1 is all the possible combinations of the different guesses for each of the bytes.

In the third phase of the attack, the attacker attempts to generate all of the possible guesses for the entire state of Round 2. Due to the "Shift Row" step of AES, the value of each of the four bytes in Round 1 affect the values of distinct four bytes in Round 2 (see Figure 12). The same guessing logic as before allows us to create the new guess list (see Algorithm 4 for example). The guess list GuessList2 is created from all the possible combinations of the guess for each four-byte group (this is done separately for each guess in GuessList1). One can repeat the same process to use GuessList2 to create the guess list GuessList3 for the state of Round 3 (and in the case of AES 256 continue another round to get GuessList4).

5.8.5 Fine Grained Cache Attack

To generate the required traces, an attacker with OS level privileges (root) monitors cache access through Prime+Probe. The attack obtains fine-grained temporal resolution through a controlled-channel attack [140] . A controlled-channel attack involves disabling the present bit on the enclave's page tables, which

by necessity are handled by the OS. By marking the page containing the T-Tables as not-present, the attacker forces an asynchronous enclave exit upon access to the table, thereby transferring control to the attacker controlled OS.

Since all of the T-Tables lie in the same page, the attacker must 'toggle' between the accesses by performing a controlled-channel attack on a page access that occurs in between each T-Table access. This work uses the page containing the topmost frame of the stack for this, as the mbedTLS implementation must first read the index into the T-Table from the stack before each access.

Unlike the T-Table addresses, however, the location of the stack is randomized by the SGX loader. This work overcomes this hurdle by first using a controlled-channel attack to force an enclave exit upon entrance to the AES function. One then marks all pages in the enclave, except for the thread control structure (TCS), saved state area (SSA), and the pages containing code, as not-present. One then resumes execution within the enclave; since the first instruction of the function prologue is `push_rbp`, control immediately returns to the custom segmentation fault handler. Within the handler, one can determine which page caused the segmentation fault, which in this case will be the page containing the top of the stack.

In this manner, one learns the location of the stack for use in the controlled-channel attack. Then, by forcing enclave exits upon each access to the T-Tables, an attacker can use a last-level cache (LCC) Prime+Probe attack to measure each T-Table access separately.

To reduce the amount of noise in the attack, the attack used Intel's cache allocation technology to partition a single way of the LLC to both the victim and attacking process, and used the `isolcpus` kernel boot parameter to isolate them on a single physical core.

Related Attacks. Roche, Lomné, and Khalfallah [174] also demonstrate a blind attack on AES. However, they consider a particularly powerful attacker who is able to generate arbitrary faults in the key schedule. Jaffe [175] attacked counter mode encryption with an unknown nonce, but required 2^{16} consecutive block

[174] Thomas Roche, Victor Lomné, and Karim Khalfallah. *CARDIS*, 2011.
[175] Josh Jaffe. *CHES*, 2007.

encryptions. Ronen et al. [176] also showed a blind attack on counter mode encryption targeting the authentication MAC.

5.8.6 Evaluation

The Victim. The experiments were performed on a Lenovo P50 laptop equipped with 16 GB of RAM and an Intel i7-6820HQ CPU clocked at 2.7GHZ with a 8MB L3 cache. The laptop was running Ubuntu 16.04.

Similar to Xiao et al. [141], this work demonstrates the viability of the differential attack against mbedTLS-SGX [123], an SGX port of the widely-used mbedTLS library. To the best of my and my co-authors' knowledge, mbedTLS-SGX is the only library currently available that features a function SGX-based HTTPS client.

Attack Procedure. This work demonstrates an end-to-end attack on a connection between the TLS client and `www.cia.gov`, with all of the client's cryptographic operations taking place within the enclave. One first mounts a Prime+Probe attack to recover the CTR_DRBG state used to generate the 256 bits of the Elliptic Curve Diffie Hellman (ECDH) ephemeral private key (a total of five AES256 encryptions of an incrementing counter). Using the recovered private key we calculated the premaster key and subsequently decrypted the HTTPS stream.

Results. Due to high noise levels in some traces, the attack successfully recovered the enclave's CTR_DRBG state in $\approx$ 36% of the 1000 trials. The online phase, during which the LCC Prime+Probe attack is mounted, takes less than two seconds. The offline phase in which the PRG state is recovered and TLS stream is decrypted took negligible time.

Attack Complexity. The complexity of the attack is dominated by calculating the set of key candidates. Generating each candidate requires $4 \cdot 2^{16}$ T-Table look-ups for each trace. Eliminating candidates by decryption required negligible work.

[176] Eyal Ronen et al. *IEEE S&P*, 2018.

Design	Certificates
CTR_DRBG	1694 (67.8%)
Hash_DRBG	906 (36.3%)
HMAC_DRBG	922 (37.0%)
Total implementations	2498

Table 10: CMVP-Certified Uses of DRBG Designs.

A further experiment tested the number of remaining candidates in each step experimentally; both in the noise free case (using a simulation over 500 random keys) and in the noisy case (1000 SGX attacks). Performing the attack with two traces yields $1.13 \cdot 2^9$ and $1.52 \cdot 2^{11}$ key candidates for the noise free and noisy cases respectively. Running the analysis with three traces immediately yields the single candidate correct in each list in the simulated environment. However, noise in the real-world setting required the provision of an extra fourth trace to narrow the analysis to a single candidate.

5.9 Impact

Database Scraping. In order to evaluate the impact of my findings, I scraped a public database of security certificates released under NIST CMVP. The results are tabulated in Tables 10–11. CTR_DRBG was the most popular design, supported by 67.8% of the implementations in the database. Of 2498 implementations present, 1694 (67.8%) supported CTR_DRBG. Of these 461 (25%) exclusively supported AES-128, 1163 (69%) supported AES-128 along with other ciphers, and 1227 (72%) supported AES-256. The CMVP database also permits modules to certify whether prediction resistance is enabled for the DRBG implementation. Of the 1694 total implementations that supported CTR_DRBG, 66 provided no information about prediction resistance, 618 supported use of the DRBG in either mode with the default unspecified, 433 explicitly enabled prediction resistance, and 577 did not support prediction resistance. Among the CTR_DRBG implementations, 85 did not use a derivation function and 1137 did not support an alternate DRBG algorithm.

Cipher	Certificates
3DES	19 (1%)
AES-128	1163 (69%)
AES-192	598 (35%)
AES-256	1227 (72%)
CTR_DRBG	1694

Table 11: CTR_DRBG Certificates. A majority of the 1694 certified implementations using CTR_DRBG use either AES-128 or AES-256. An implementation may support more than one of these modes.

5.10 Discussion

Limitations. The results in this chapter rely on a victim's use of T-Table AES, which has long been known to leak information via side-channels. However, as illustrated in this work T-Table AES is still used by many modern implementations. In the non-Software Guard Extensions (SGX) setting, the TLS attack requires code execution on the client, and further succeeds only after thousands of handshakes. This potentially allows for detection of an on-going attack. While this work demonstrates the SGX attack against the only library that provides a working end-to-end example of an HTTPS client, the Intel-supported SGX-SSL cryptographic library [177] (which does not provide support for TLS) uses SGX's hardware-based `RDRAND` PRG and therefore is not vulnerable to a T-Table based attack.

The Fragility of ECDSA. The fragility of DSA and ECDSA in the face of random number generation and implementation flaws has been repeatedly demonstrated in the literature [178,179]. It is inevitable that a random number generation failure would compromise a single session or a signature, but DSA/ECDSA are particularly vulnerable to compromise of long-term secrets. Deterministic ECDSA, defined in RFC6979 [180], is the recommended countermeasure.

[177] Intel. *Intel Software Guard Extensions SSL*. Online: https://github.com/intel/intel-sgx-ssl. 2017
[178] Joachim Breitner and Nadia Heninger. *FC*, 2019.
[179] Yuval Yarom and Naomi Benger. *ePrint*, 2014.
[180] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. Internet Engineering Task Force, Aug. 2013

CHAPTER 6

FROM THE ASHES

This chapter provides a comparative analysis of the pseudorandom number generators (PRGs) assessed in this work, and outlines a series of recommendations for random number generation for system implementers, forced to choose today between available designs and implementations.

> All information looks like noise until you break the code.
>
> *Neal Stephenson*

6.1 Comparing Attacks

Juniper in Context The attack surface of Dual EC as described in Chapter 4 is similar to the attack on X9.31 described in Chapter 3, 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. In contrast, 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.[xxviii] This second restriction plays a major role in the cost of an attack on a protocol such as Transport Layer Security (TLS) or Internet Protocol Security (IPsec).

The choice by the Dual EC attacker to target the random number generation subsystem is instructive. Random number generators have long been discussed in theory as a target for kleptographic substitution attacks [181], but this incident tells us that the threat is more real than indicated by the academic literature.

From the perspective of an attacker, by far the most attractive feature of the ScreenOS PRG attack is the ability to significantly undermine the security of ScreenOS *without* producing any externally-detectable indication that would mark the ScreenOS devices as vulnerable. This is in contrast to previous well-known PRG failures, which were externally observable, and, in the case of the Debian PRG flaw, actually detected through observational testing. Indeed, the versions of ScreenOS containing an attacker-supplied parameter appear to have produced output that was cryptographically indistinguishable from the output of previous versions, thus preventing any testing or measurement from discovering the issue.

Ease of X9.31 Exploitation The attack against the X9.31 PRG was significantly less computationally expensive to carry out than many of the attacks against Dual EC in TLS measured in Chapter 4. This is because the most efficient attacks against Dual EC require 32 bytes of raw PRG output, and the effort

[xxviii] 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.

[181] Adam Young and Moti Yung. *EUROCRYPT*, 1997.

PRG	Last Valid	Primitive	Attacks	Security Proof	Flaw
X9.31	2016	AES/3DES	Passive[a]	—	No Keying Advice
Dual_EC	2014	Elliptic Curves	Passive[b]	[78]	Choice of parameters
CTR_DRBG	c	AES/3DES CTR	Active[c]	—	Entropy optional, Lacking Threat Model
HMAC_DRBG	c	Keyed Hash	Theoretical	[18][d]	Entropy optional
Hash_DRBG	c	Hash Function	—	[18]	—

[a] Chapter 3
[b] Chapter 4
[c] Chapter 5
[d] Woodage and Shumow showed that HMAC_DRBG is not forward secure if sufficient additional entropy is not included with every call to generate.

Table 12: Comparison of Analyzed Attacks. In this table, c indicates that a design is still currently approved for use.

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. As described in Section 4.5.2, when Juniper replaced the X9.31 PRG with Dual EC in their ScreenOS implementation, they increased the length of the nonces used in the Internet Key Exchange (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.

SP 800-90A SP 800-90A was purportedly an attempt to get random number generation 'right'. The attack on CTR_DRBG, and the abject failure of the Dual EC standard indicates that this goal was not met. The further weakness in HMAC_DRBG discovered by Woodage and Shumow [18] leaves only one of the original four PRGs in the standard as a safe-by-default choice. While the CTR_DRBG attack required a fairly powerful attacker, its lesson on threat modelling should not be taken lightly.

[xxix] Of course, reducing nonce size cannot prevent all data exfiltration strategies. However, it may increase the difficulty of hiding the necessary code, and the complexity of executing an attack.

6.2 Lessons From PRGs

6.2.1 PRG Output & Protocol Design

Allowing nonces to vary in length, and in particular to be larger than necessary for uniquely identifying sessions, is a bad idea. I and my co-authors on the Juniper work are unaware of any cryptographic rationale for 256-byte nonces, as permitted by IPsec; it is simply an invitation for implementations to disclose sensitive state, intentionally or not.[xxix]

Adding even low-entropy shared secrets as key derivation inputs helps protect against entropy failures. One observes a difference in exploitability of the ScreenOS bugs between IKEv1 and IKEv2 that is entirely due to the different use of the PSK between the two protocols. It is unfortunate that IKEv2 is easier to exploit.

All the attacks developed in this work rely on an attacker's access to the raw output from the PRG. While implementers can certainly mitigate these attacks by passing PRG output through a one-way function, protocol designers may additionally wish to hedge their designs against entropy failure [182]. There are few widely deployed cryptographic protocols but many crypto libraries and randomness subsystems, so changes to protocols may do more good than revised design guidelines for PRGs. One approach is to *derandomize* protocols by replacing randomized nonces with deterministic, cryptographically secure values derived from connection parameters, to avoid nonce collisions or predictability due to improperly seeded random number generators. Bellare, Paterson, and Rogaway [20] propose techniques to render symmetric cryptographic primitives deterministic and resilient against *algorithm substitution attacks*. Future research might investigate whether similar guarantees can be extended to protocols employing asymmetric primitives such as key exchange protocols.

For implementers Cryptographic code must be locally auditable: It must be written in such a way that examining a function or a module in isolation allows the reader to understand its behavior.

ScreenOS's implementation failed to live up to this guideline. A loop counter in the core `prng_generate` routine was defined as a global variable and changed in a subroutine. This is a surprising-enough pattern

[182] Mihir Bellare et al. *ASIACRYPT*, 2009.

that several experienced researchers who knew that the routine likely had a bug failed to spot it before Willem Pinckaers' contribution. The `prng_generate` routine and the `prng_reseed` routine reuse the same 32-byte buffer, `prng_temporary`, for two entirely different purposes: Dual EC output with which to seed X9.31, and output from the PRG subsystem. ScreenOS's use of pregeneration queues makes it difficult to determine whether nonces or Diffie–Hellman shares are generated first. Someone reading the code for the top-level functions implementing IKE in isolation will conclude that Diffie–Hellman shares are generated first, whereas in practice the opposite is usually the case.

The state recovery attacks suffered by Juniper suggest that implementations may wish to avoid revealing the raw output of a random number generator entirely, perhaps by hashing any PRG output before using it as a nonce. One could also design implementations so that separate PRGs are used for different protocol components, to separate nonce security from key security.

Several of the above mistakes represent poor software engineering practices. Cryptographic code reviews, whether internal or external (e.g., for FIPS validation), should take code quality into account.

6.3 Generating Random Numbers Today

The past decade has seen considerable progress in the theoretical design of secure PRGs [9,12,13,22]. Further research has produced strategies for retaining certain security properties in light of backdoors by using "immunization" strategies [22,183]. However few, if any, of these designs have seen widespread deployment. As a result, my recommendations for selecting a PRG are suboptimal.

Using `RDRAND` without a PRG. Using the built-in CPU PRG to mitigate concerns with software PRGs is not a panacea. In the several Software Guard Extensions (SGX) ports reviewed herein (including Intel's official port for OpenSSL [177]) the software PRG was replaced with calls to the `RDRAND` instruction. While using the CPU's generator avoids software side-channels, the existence of hard-to-discover bugs

[183] M. Fischlin, C. Janson, and S. Mazaheri. *CSF*, 2018.

in PRGs integrated into CPUs [184,185] mean this feature is better used as one of many sources of entropy for a provably secure software PRG.

Using the Linux PRG The Linux kernel PRG (`/dev/random` and `/dev/urandom`) until 2016 used secure hashing as the basic for its design. Unfortunately, the historical design featured significant (unnecessary) complexity in the form of an entropy estimator tied to a pooled entropy source that frustrated formal analysis [9]. While the kernel supports the inclusion of entropy from many sources, including hardware random number generators, multiple attacks on the generator have exploited the mixing, estimation and pool mechanisms [9,32,186]. The design included in kernels 4.8 and later substitutes the hashing design for one based on the ChaCha20 stream cipher, but still suffers from the same seeding problems. Discussions among leading kernel implementors are continuing in an effort to address some of the concerns around the API and seeding structure of the PRG [187].

FIPS Compliance Of the approved PRG designs in NIST SP 800-90A, Hash_DRBG appears to be on sound footing, both in practice and in theory. While this claim is supported by robustness proof in Woodage and Shumow [18], Hash_DRBG unfortunately trails CTR_DRBG significantly in number of approved modules, suggesting that there is considerable room for developers to improve on their selection of PRG.

Using CTR_DRBG CTR_DRBG's flaws, both theoretical and practical, suggest that implementations need to take great care when choosing this design. 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 CTR_DRBG [5] 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

[184] Wtdrog. *Systemd Issue #11810 - Can't suspend again after suspending one time*. May 2019

[185] Hin-Tak Leung. *Redhat Bug 1150286 - rdrand instruction fails after resume on AMD CPU*. Online: https://bugzilla.redhat.com/show_bug.cgi?id=1150286. 2019

[186] Daniel J. Bernstein. *Entropy Attacks!* Online: http://blog.cr.yp.to/20140205-entropy.html. 2014

[187] Jake Edge. *Fixing getrandom()*. Online: https://lwn.net/Articles/800509/. LWN.net, 2019

indistinguishable from a correct implementation [18]. Moreover, such vulnerabilities might not be visible in test modes due to implementation differences [188].

Where CTR_DRBG cannot be replaced, implementers should use AES hardware instructions, limit the quantity of data that can be requested in a single call, reseed frequently, and populate *addin* with high quality entropy, to provide defense in depth against the attacks in this work. In general, constant-time code should be used for all cryptographic applications, unless hardware support (e.g., AES-NI) is available.

Overall Recommendation for non-Cryptographers

The analysis of this work suggests the a strong overall choice for a software based PRG is Hash_DRBG, a design that has been adopted commercially and is available in various operating system kernels and cryptographic libraries. This design is FIPS compliant, providing a strong choice for entities operating in regulated industries.

Despite the strengths of this design, care must be taken to ensure that as with any PRG, that sufficient entropy is provided to the instantiation and that reseeding occurs regularly.

6.4 Standardizing PRGs

Juniper followed then-current best practices in designing and verifying their random number generators. They used a NIST-certified algorithm, followed the FIPS-recommended procedure to verify the output using test vectors, and followed a commonly-recommended engineering guideline to use a PRG as a whitener for a potentially insecure random number generator.

In this case, all three approaches failed. In particular, a crippling defect in the whitening countermeasure managed to go undetected in FIPS certification. This suggests potential future work for research in the

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

verification of cryptographic systems. One step would be to track the origin and use of any buffers — especially shared buffers — and enforce a rule that all random number generator output can be traced back to an appropriate cryptographic function, such as a block cipher or hash.

To the extent that FIPS guidelines mandate the use of global state, they run counter to the suggestion above, that cryptographic code be locally auditable.

Products are evaluated against FIPS standards by accredited laboratories. ScreenOS was FIPS certified with the X9.31 PRG, yet the lab evaluating ScreenOS failed to spot that X9.31 was never invoked, as well as failing to detect the defect in the Dual EC implementation described in Section 4.4. NIST should revisit its laboratory accreditation program to ensure more thorough audits, especially of randomness subsystem code.

CHAPTER 7

PSYCHOS

This chapter introduces pre-supply chain operations (PSYCHO), describing a threat model encompassing an adversary with the power to corrupt standardization and drive implementation. I review a number of standardization processes under question, culminating in recent attempts by the NSA to standardize Simon and Speck at the International Standards Organization (ISO).

Everything is foreseen; yet free will is given.

Pirkei Avot 3:15



Contributions

The chapter draws briefly on work from *"Measuring small subgroup attacks against Diffie–Hellman"* coauthored with Luke Valenta, David Adrian, Antonio Sanso, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. My contributions to that paper were in assessing implementations for the support of small Diffie–Hellman subgroups, and subsequent discovery of a partially exploitable implementation.

It also references *"DROWN: Breaking TLS using SSLv2"* by Nimrod Aviram *et al.*. My contribution to that work was in maintaining cryptanalytic infrastructure and code used to execute the attack.

- I develop and conceptualize adversarial use of *pre-supply chain operations*, describing how an adversary can encourage the standardization of design flaws, and attempt to ensure that implementations both contain the flaw, and are exploitable.
- Drawing on the prior chapters, I outline strategies that such an attacker may use, reviewing the history of attempts to manipulate standardization, focusing on suspicions of malfeasance.
- I claim that the third-party testing lab model does not result in improvements to security, and to the contrary provides a false sense of security. Further, I claim this disconnect between standards and security is a broader symptom of U.S. Federal cryptographic standards.
- I suggest that replacing this model with automated testing of pseudorandom number generators (PRGs) analysis tooling would be a step forward, but that a superior solution would be the use of formally verified implementations, based on PRGs that are provably secure under robust models.

7.1 Snowden Redux

In 2011 Edward Snowden leaked a large number of classified documents to Der Speigel and The Guardian. These documents intimated that the NSA possesses passive decryption capabilities against large fractions of Internet traffic [16]. Academics and journalists have proposed numerous explanations for these capabilities [21,25,93,189]. Explanations that don't involve PRG exploitation include precomputation attacks against Diffie–Hellman key exchanges, such as those outlined in Adrian et al. [93] and use of malware 'implantation' [189,190]. However, the suspicious nature of the Dual EC standardization process, along with slides marked as part of the 'BULLRUN' NSA program and with associated reporting, suggest that U.S. intelligence agencies used a combination of methods to compromise cryptographic deployments. The Dual EC flaw (Chapter 4) and the X9.31 flaw (Chapter 3) may account for some fraction of the alluded to capabilities.

These events are instructive in understanding standardization based attacks, and will inform discussions within this chapter.

[189] Sean Gallagher. *Photos of an NSA "upgrade" factory show Cisco router getting implant*. 2014
[190] Jacob Appelbaum, Judith Horchert, and Christian Stöcker. , 2013.



Figure 13: Clay Tablet with inscription of Annals of Tiglath-Pileser III. This Neo-Assyrian tablet contains early references to the destruction of supply chains. The tablet reads: "I cut down the orchards (and) trees that were near his city wall; I did not leave a single one. I killed date-palms throughout his land by ripping off their fruit and filling the meadows with them. I destroyed, devastated, and burned with fire all of his cities."

7.2 Expanding Threat Models

Supply Chain Attacks are adversarial actions that (a) are surreptitious; (b) leverage broader procurement and deployment ecosystem to carry out the attack; (c) and are outside the immediate control of the administrator of the resource[xxx].

While threats to supply chains themselves have been a source of commentary since ancient times [191] (see Figure 13), surreptitious hijacking of supply chains for malicious purposes appears to be a more recent phenomenon. In this decade, discoveries of state-sponsored supply chain attacks on computerized resources motivate a renewed look at threat models.

In 2013, MITRE Corporation (on behalf of the Office of the Assistant Secretary of Defense for Systems Engineering) developed a comprehensive framework for the assessment of supply chain threats. As part

[xxx] In the field of information assurance, supply chain attacks are so-named for how they exploit the supply chain, not because the target is the supply chain itself.

[191] Jacob L. Wright. *Journal of Biblical Literature*, 2008.

of that project, Reed, Miller, and Popick [192] published a technical report identifying forty-one different types of supply chain attacks. I include a graphic from the report outlining their model in Figure 14.

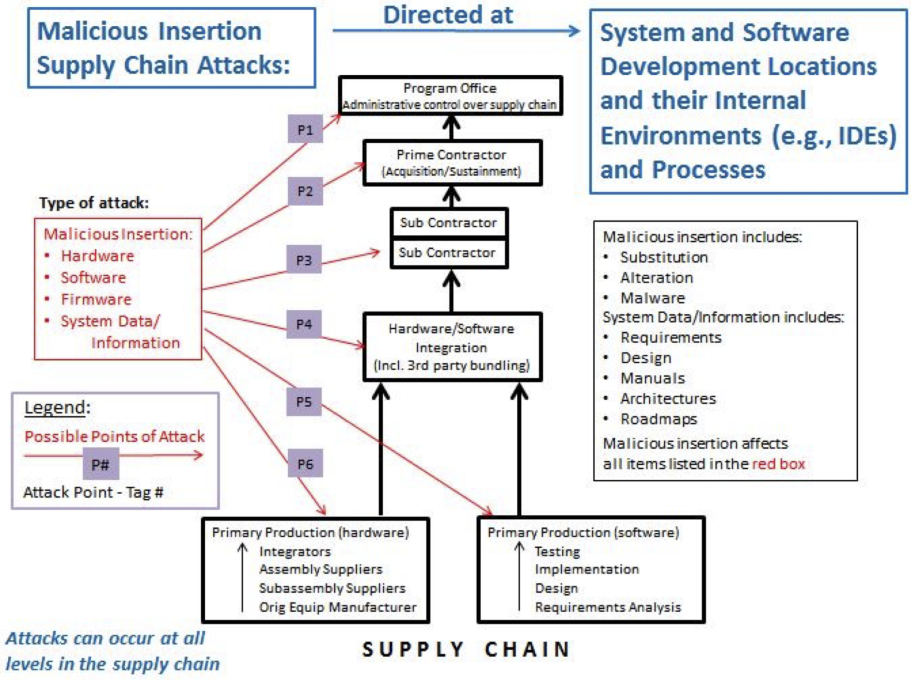


Figure 14: MITRE Corporation Supply Chain Attack Model.

All the attacks considered by the report ultimately rely on a mismatch between a specification, assumed to be correct, and an accessed resource that has been compromised in the intervening development and deployment process. This is representative of the prior state-of-the-art in evaluating supply chain risks.

[192] Melinda Reed, John F Miller, and Paul Popick. *Supply chain attack patterns: Framework and Catalog*. Tech. rep. 2013

Pre-Supply Chain Attacks

Traditional Supply Chain Attack models have generally assumed a trusted domestic regulatory and legal framework [193], have focused on counterfeiting risks [194], or are the result of modelling on behalf of the government [195] (thus excluding the government itself from the set of untrusted actors). Past work has also concentrated analysis on cyber physical systems, with little discernable academic work focussing on cryptographic applications.

Models including those of Reed, Miller, and Popick [192] and DiMase et al. [193] begin their analysis after the initial design requirements and specifications have been set, excluding early development phases.

While the models designed for the Department of Defense might reasonably exclude certain state actions from consideration, in a world where non-U.S. designs are increasingly considered in international standards bodies, a model incorporating state actors is of growing import.

As a preliminary attempt at categorizing PSYCHOs, I propose the following set of techniques a state adversary may use to compromise cryptographic primitives, protocols, and implementations (all of which are herein referred to as 'cryptosystems'):

[193] Daniel DiMase et al. *Environment Systems and Decisions*, 2015.
[194] Bhanu Sood, Diganta Das, and Michael Pecht. , 2011.
[195] Dan Kaufman. *An analytical framework for cyber security*. Tech. rep. Defense Advanced Research Projects Agency, 2011

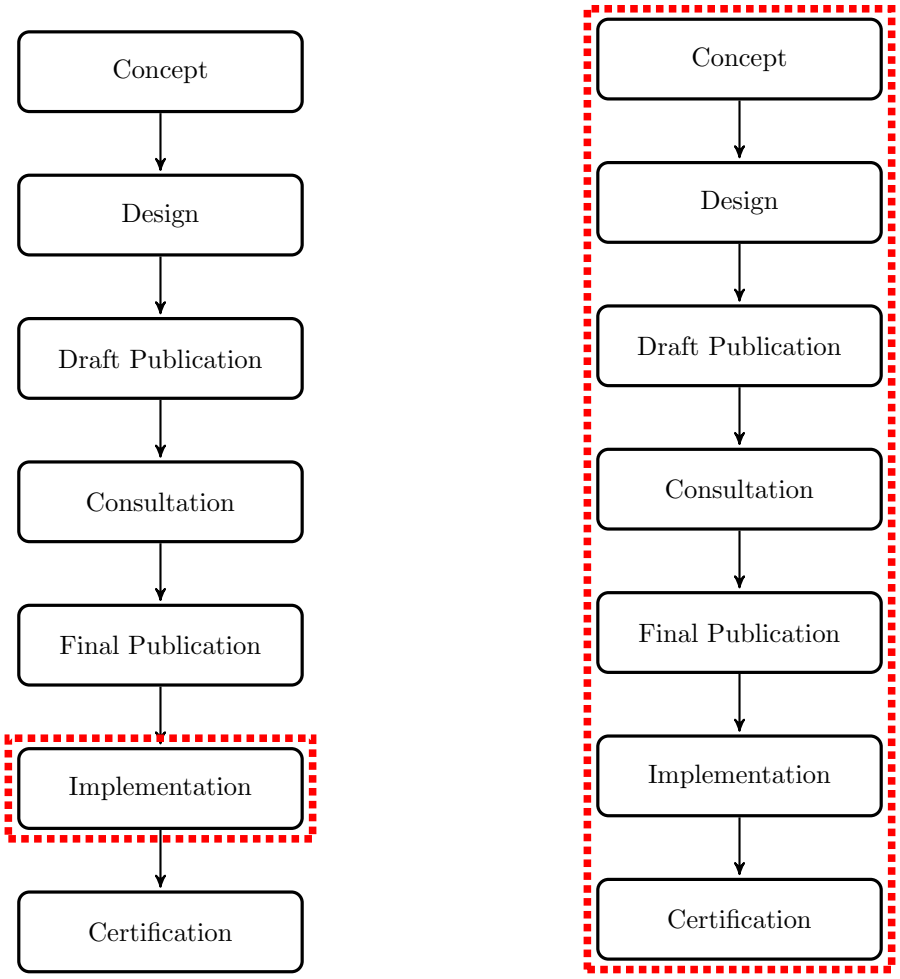
Taxonomy of PSYCHOs

- **Regulatory Imposition.** An adversary imposes regulations with the intent on precipitating sub-optimal design decisions (beyond the direct impact of the regulation). Eg; Export restrictions, key escrow requirements that introduce fragility beyond the facially apparent weakness.
- **Design Flaw.** A cryptosystem is designed with a subtle flaw, that if noticed, is minimized, and otherwise may escape attention. This includes omissions of guidance strictly necessary for safely implementing an otherwise secure design.
- **Parameter Manipulation.** A cryptosystem is standardized for use with fixed parameters that introduce a vulnerability. In this model, I exclude parameters internal to the cryptographic primitives (such as choice of SBox), referring to parameters exposed to either protocol and standards designers, or implementers.
- **Intentional Complexity.** A cryptosystem is rendered so complex as to increase the likelihood that implementors introduce bugs, leading to exploitability.
- **Threat Model Exclusion.** A standard is composed, but excludes a significant class of vulnerabilities from its threat model, leading implementors to neglect countermeasures.

As shown in Section 7.5, assignment of an attack to one category does not exclude secondary impact of other categories, as there is substantial overlap and the model serves to provide context rather than act as a rigid definitional tool.

The model also assumes a particularly powerful attacker—a nation state armed not only with sophisticated cryptanalytic capabilities and a vast budget, but the ability to influence policy making and the institutional gravitas to sway standards bodies. While the number of such actors are limited, the body of work presented herein suggests that the threats to cryptographic secured posed by such actors are very real.

Interestingly, given the human-centric nature of many standardization processes, psychological operations as traditionally carried out by intelligence agencies may prove effective in the tactical level execution of a PSYCHO. By a limited manipulation of key decision makers or groups within standards organizations,



(a) Supply Chain Attacks.

(b) Pre-Supply Chain Attacks.

Figure 15: Comparing Supply Chain Attacks and PSYCHOs. Traditional supply chain attacks assume a trusted preliminary conceptualization and design process, and following that, a standardization process that addresses both operational and technical needs. The PSYCHO model broadens the set of considered adversaries, viewing the early pre-supply chain processes as corruptable by an actor with sufficient leverage with the relevant institutions.

an adversary may be able to leverage influence operations targeting a small number of individuals into mass decryption capabilities. The magnifying impact of corrupting a standard highlights the potential power of this attack.

I now turn to applying the model to a series of known instances of standardization failure, assessing the fit between the model and the events described.

7.3 Dangerous Defaults

Diffie–Hellman key exchange is one of the most common public-key cryptographic methods in use in the Internet. It is a fundamental building block for IPsec, SSH, and TLS. In the textbook presentation of finite field Diffie–Hellman, Alice and Bob agree on a large prime p and an integer g modulo p. Alice chooses a secret integer x_a and transmits a public value $g^{x_a} \mod p$; Bob chooses a secret integer x_b and transmits his public value $g^{x_b} \mod p$. Both Alice and Bob can reconstruct a shared secret $g^{x_a x_b} \mod p$, but the best known way for a passive eavesdropper to reconstruct this secret is to compute the discrete log of either Alice or Bob's public value. Specifically, given g, p, and $g^x \mod p$, an attacker must calculate x.

In order for the discrete log problem to be hard, Diffie–Hellman parameters must be chosen carefully. A typical recommendation is that p should be a "safe" prime, that is, that $p = 2q + 1$ for some prime q, and that g should generate the group of order q modulo p. For p that are not safe, the group order q can be much smaller than p. For security, q must still be large enough to thwart known attacks, which for prime q run in time $O(\sqrt{q})$. A common parameter choice is to use a 160-bit q with a 1024-bit p or a 224-bit q with a 2048-bit p, to match the security level under different cryptanalytic attacks. Diffie–Hellman parameters with p and q of these sizes were suggested for use and standardized in DSA signatures [73]. I here refer to these non-safe primes as DSA primes, and to groups using DSA primes with smaller values of q as DSA groups.

7.3.1 Small Subgroup Attacks

A downside of using DSA primes instead of safe primes for Diffie–Hellman is that implementations must perform additional validation checks to ensure the key exchange values they receive from the other party are contained in the correct subgroup modulo p. The validation consists of performing an extra exponentiation step. If implementations fail to validate, a 1997 attack of Lim and Lee [196] can allow an attacker to recover a static exponent by repeatedly sending key exchange values that are in very small subgroups. In "Measuring small subgroup attacks against Diffie–Hellman" [197] my co-authors and I describe several variants of small subgroup confinement attacks that allow an attacker

[196] Chae Hoon Lim and Pil Joong Lee. *CRYPTO*, 1997.
[197] Luke Valenta et al. *NDSS*, 2016.

with access to authentication secrets to mount a much more efficient man-in-the-middle attack against clients and servers that do not validate group orders. Despite the risks posed by these well-known attacks on DSA groups, NIST SP 800-56A, "Recommendations for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography" [198] specifically recommends DSA group parameters for Diffie–Hellman, rather than recommending using safe primes. RFC 5114 [199] includes several DSA groups for use in IETF standards.

My co-authors and I observed that few Diffie–Hellman implementations actually validate subgroup orders, in spite of the fact that small subgroup attacks and countermeasures are well-known and specified in every standard suggesting the use of DSA groups for Diffie–Hellman, and DSA groups are commonly implemented and supported in popular protocols. For some protocols, including TLS and SSH, that enable the server to unilaterally specify the group used for key exchange, this validation step is not possible for clients to perform with DSA primes—there is no way for the server to communicate to the client the intended order of the group. Many standards that use DSA groups further suggest that the order of the subgroup should be matched to the length of the private exponent. Using shorter private exponents yields faster exponentiation times, and is a commonly implemented optimization. However, these standards provide no security justification for decreasing the size of the subgroup to match the size of the exponents, rather than using as large a subgroup as possible.

RFC 5114 Design Rationale Neither NIST SP 800-56A nor RFC 5114 give a technical justification for fixing a much smaller subgroup order than the prime size. Using a shorter private exponent comes with performance benefits. However, there are no known attacks that would render a short exponent used with a safe prime less secure than an equivalently-sized exponent used with in a subgroup with order matched to the exponent length. The cryptanalysis of both short exponents and small subgroups are decades old.

[198] Elaine B Barker, Don Johnson, and Miles E Smid. *SP 800-56A. recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)*. Tech. rep. 2007

[199] M Lepinski and Steve Kent. *Additional Diffie-Hellman Groups for Use with IETF Standards*. RFC 5114. Internet Engineering Task Force, 2008

If anything, the need to perform an additional modular exponentiation to validate subgroup order makes Diffie–Hellman over DSA groups *more* expensive than the safe prime case, for identical exponent lengths. As a more minor effect, a number field sieve-based cryptanalytic attack against a DSA prime is computationally slightly easier than against a safe prime. The design rationale may have its roots in preferring to implicitly use the assumption that Diffie–Hellman is secure for a small prime-order subgroup without conditions on exponent length, rather than assuming Diffie–Hellman with short exponents is secure inside a group of much larger order. Alternatively, this insistence may stem from the fact that the security of DSA digital signatures requires the secret exponent to be uniformly random, although no such analogous attacks are known for Diffie–Hellman key exchange [200].

7.3.2 The Logjam Attack

In 2015, a diverse team of researchers noted that Diffie–Hellman key exchange is commonly performed over the same standardized or otherwise fixed algebraic groups [93]. These groups are used within many critical networking protocols including TLS, IKE, SSH and Tor. The researches noted that the Number Field Sieve (NFS) algorithm used to compute discrete logarithms (and thus a Diffie–Hellman handshake's security) begins with a step dependent not on the individual discrete log, but rather on the group. This precomputation step is the bulk of the computational work required, with the individualized 'descent' step taking a trivial amount of time and work by comparison. While reuse of algebraic groups is facially benign, should an adversary perform precomputation for a highly reused group she would then be able to amortize the cost of breaking handshakes, spending little extra work per each additional handshake.

The aforementioned 'Logjam' attack paper focuses on a TLS cipher suite downgrade attack, forcing the handshake to use a 512-bit group, for which the precomputation step is tractable for an academic computing cluster. Going beyond academic efforts, the researchers develop an argument that not only was precomputation over 1024-bit groups feasible for the National Security Agency (NSA), classified slides leaked by Edward Snowed add plausibility to the claim that the agency was actively exploiting this vector.

[200] Phong Q Nguyen and Igor E Shparlinski. *Designs, codes and cryptography*, 2003.

7.3.3 Diffie–Hellman PSYCHOs

Elements of both the RFC 5114 and Logjam narratives comport with the PSYCHO model advanced by this work.

The Logjam attack is a product of multiple factors, some of which correspond to PSYCHO vectors: regulatory impositions that weaken standards, and unnecessary introduced complexity. While the complexity of SSL/TLS is not wholly a product of known government action, the combination of cipher suite agility and existing weak ciphers rendered the protocol vulnerable.

Adopting the Diffie–Hellman group recommendations from RFC 5114 and NIST SP 800-56A creates vulnerabilities for organizations using existing cryptographic implementations, as many libraries allow user-configurable groups but have unsafe default behaviors. While ideally a standards body would create a user-friendly standard and build bulwarks again implementation fragility when designing or updating cryptographic standards, a sophisticated adversary might see these difficulties as opportunity for a PSYCHO. The lacking justification for the RFC 5114 and NIST SP 800-56A group recommendations fits neatly into this model.

7.3.4 Export Cryptography, Complexity, and Agility

In 1993 the U.S. Bureau of Industry and Security published a revised set of export restrictions to limit the dissemination and use of cryptography with keys above certain lengths. The limitations included:

- Employing a symmetric algorithm with a key length in excess of 56-bits;
- Employing an asymmetric algorithm based on:
  - A factorization of integers in excess of 512 bits (i.e. RSA);
  - Computation of discrete logarithms in a multiplicative group of a finite field of size greater than 512 bits (i.e. Diffie–Hellman over Z/pZ);
  - Discrete logarithms in a group in excess of 112 bits (i.e. Diffie–Hellman over an elliptic curve);

These restrictions were ostensibly designed to ensure that the breaking of exported cryptography was within the cryptanalytic capabilities of the U.S. intelligence community of the time.

As a result of these restrictions, contemporaneously developed protocols often featured modes compliant with these restrictions. In particular, SSL/TLS included a set of `EXPORT` cipher suites that limited key lengths in compliance with the requirements. While the design was intended to allow domestic parties the benefit of full key length security, a significant line of research over the past decade located a substantial number of mixed protocol, implementation, and cryptographic flaws in SSL/TLS that targeted the stronger suites. A subset of these attacks including DROWN [201], POODLE [202] and FREAK [203] relied on support for legacy "export grade" ciphers.

The burden imposed by adhering to the regulations not only opened up attacks on the `EXPORT` cipher suites, but on the overall protocols which were forced to include complex mechanisms to switch between strong and weak cipher-suites depending on client and server characteristics.

[201] Nimrod Aviram et al. *USENIX Security*, 2016.
[202] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. *This POODLE bites: exploiting the SSL 3.0 fallback*. Tech. rep. 2014
[203] B. Beurdouche et al. *IEEE S&P*, 2015.

7.4 A Failed PSYCHO?

The descriptive portion of this section draws heavily from the work of Tomer Ashur, who was involved directly in the events here recounted as a representative of the Belgian Delegation to the ISO. In particular, the retelling draws from a talk at Real World Crypto (2019) given by his co-author Atul Luykx, entitled "An Account of the ISO/IEC Standardization of Simon and Speck" [204].

ISO. Beyond the NIST standards, and outside of the U.S, the International Organization for Standardization (ISO) oversees many cryptographic standards. The ISO is comprised of national standards bodies, each of whom represent their country of origin. The U.S. represents its interest through American National Standards Institute (ANSI), the same body responsible for the X9.31 PRG discussed in Chapter 3. The major benefit of ISO standardization is promotion of trade; the World Trade Organization (WTO) encourages national regulators to base domestic measures on international standards, thereby reducing barriers to trade. As a result of the nation-based representation and voting structure of the organization, international relations are known to play a significant role in determining the outcome of particular standards. Notably, decision-making within the organization is based on an ill-defined notion of consensus, starting with subject matter experts during the study and working group periods, to nation-level consensus at the committee stage and onward.

SIMON & SPECK. In June 2013, the NSA published specifications for two cipher families: SIMON and SPECK, optimized for lightweight hardware and software implementations respectively. According to the NSA, the publication was prompted by concerns that existing ciphers were too heavyweight for the Internet of Things and were insufficiently interoperable[205]. In October 2014, less than six months after the Snowden revelations, the U.S. delegation submitted the ciphers to the ISO for consideration.

Failed Standardization. Within the ISO, the group with oversight responsibility over cryptography, and therefore over the standardization of Simon and Speck is ISO/IEC JTC 1/SC 27/WG 2[xxxi]. During

[xxxi] Joint Technical Commitee (JTC) 1 oversees Information Technology standards, Subcommittee (SC) 27 is responsible for Security, and finally, Working Group 2 is responsible for Cryptography.

[204] Tomer Ashur and Atul Luykx. *RWC*, 2019.
[205] National Security Agency. *Algorithms to Support the Evolution of Information Assurance Needs*. Tech. rep.

the JTC 1/SC 27/WG 2 standardization process Tomer Ashur, along with other delegation members, raised a number of concerns, in particular, that the proposal lacked a design rationale for key features including a series of matrices central to the cipher [206,207].

Following skepticism and opposition from Tomer Ashur and others, the NSA released "Notes on the design and analysis of SIMON and SPECK," [208] purporting to provide the transparency and design rational desired.

In an email to the 'Linux Crypto' mailing list Ashur outlined the following troubling elements of the NSA's representations to the ISO to public researchers:

- Omissions in the design rationale pertaining to decisions such as the number of rounds, the key schedule, and the U, V and W matrices used in Simon's design.
- Misleading statements as to the strength of some of their proposed mitigations, claiming that cited works represented the best possible attacks.
- Falsehoods or severe errors that limited the types of attacks the authors were willing to consider and misrepresented the strength of existing attacks.

The NSA further refused to answer questions to the satisfaction of the other JTC 1/SC 27/WG 2 participants and argued that they should be trusted as one of the most prominent employers of expert cryptographers. In a written response to concerns the NSA also claimed that the ISO document was not an appropriate forum for discussion as to the motivation for the U, V and W matrix designs, but proposed no alternative venue. As I discuss in Section 7.6.3, the public's faith in the NSA was at a low following the events of the first Crypto Wars and the Snowden disclosures and thus these argument retained little cachet. As Komninos noted in an interview with Computer Business Reviews[209]: "Unfortunately the NSA lacks a good reputation in academia.... Bottom line is that in most cases, the NSA has pushed for adoptions/standards in which security vulnerabilities were found very soon after."

[206] Tomer Ashur. *[PATCH v2 0/5] crypto: Speck support*. Linux Crypto, Mailing List. 2018
[207] Tomer Ashur and Atul Luykx. *EUROCRYPT Rump Session*, 2018.
[208] Ray Beaulieu et al. *Notes on the design and analysis of SIMON and SPECK.*. 2017
[209] Ed Targett. *NSA Ciphers "Simon and Speck" Are Dead – But Not Entirely Buried Says ISO*. Online: https://www.cbronline.com/news/nsa-ciphers-iso. Computer Business Review, 2018

While some 'procedural maneuvers' [204] allowed the proposal to advance through multiple rounds of review, but were ultimately unsuccessful at the second stage of the recognition processes wherein technical experts (rather than nation-state delegates) are required to weigh-in [209].

Use in Popular Software. In mid-2018, Google Employees Eric Biggens and Theodore T'so (the `/dev/random` maintainer), merged Speck into the Linux Kernel[xxxii] for use in filesystem encryption [211]. T'so claimed the cipher was added to support "lowest-end Android devices".

The inclusion of Speck within the Linux Kernel resulted in substantial backlash [212,213] that eventually led to Google withdrawing support for the change. Within three months a patch was merged, eliminating the cipher [214]. T'so responded to the controversy stating "Let's be clear — the arguments about whether or not to use Speck, and whether or not to remove Speck from the kernel, are purely political — not techinical [sic]."[215]

Fitting the Model. The circumstances surrounding the standardization of Simon and Speck fit the PSYCHO model: a powerful entity (the NSA) used its institutional weight to cause the (near) adoption of a standard that would not have been adopted absent the substantial power imbalance and further appeared to have done so with possible ill-motive. Though the actual strength of the algorithms remain unknown, the model advanced in this thesis suggests that cryptographers were right to be wary.

[xxxii] Changes were added in both the kernel version 4.17 and 4.18 merge windows[210].

[211] Michael Larabel. *EXT4/Fscrypt Changes For Linux 4.18: Speck File-System Encryption Being Added*. Online: https://www.phoronix.com/scan.php?page=news_item&px=EXT4-Fscrypt-Speck-Crypto-4.18. 2018

[212] jean-michel. *ALERT! Linux kernel 4.17 contains a backdoor.* Obarun Forum, Online: https://forum.obarun.org/viewtopic.php?id=584. 2018

[213] Abhishek Prakash. *NSA's Encryption Algorithm in Linux Kernel is Creating Unease in the Community*. 2018

[214] Jason A. Donenfeld. *crypto: speck - remove Speck*. Online: https://git.kernel.org/pub/scm/linux/kernel/git/herbert/cryptodev-2.6.git/commit/?anzwix=1&id=578bdaabd015b9b164842c3e8ace9802f38e7ecc. 2018

[215] Theodore T'so. *Re: [PATCH] crypto: remove speck*. Online: https://lkml.org/lkml/2018/8/7/30. 2018

[210] Eric Biggers. *crypto: speck - add support for the Speck block cipher*. Online: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da7a0ab5b4babbe5d7a46f852582be06a00a28f0. 2018

7.5 Contextualizing the Attacks

While the attacks developed in Chapters 3–5 may be facially distinct from the non-PRG based cryptographic attacks surveyed in this chapter, they are all joined by the common thread of concerns pertaining to their respective design and standardization processes.

In Table 13 I depict the set of attacks discussed in this work, alongside their PSYCHO categorization. While the application of the model to an attack *does not* constitute a definitive claim that any particularly entity was responsible for a corresponding PSYCHO, it does either

1. Create reasonable grounds for suspicion (where media reporting, leaked documents, or other evidence suggests foul-play)
2. Provide an exemplar of how an adversary might conceivably attack a cryptosystem.

Each of the attacks covered by this work can be viewed through the lens of the PSYCHO model.

The X9.31 attack represents the simplest example—creating a vulnerability by leaving a gap in the design and associated standard. A PSYCHO attack using X9.31 would have had long lasting impact as the design retained its standardization status for decades, and was only sunset as part of a broad modernization effort. While serious flaws in Simon and Speck are yet to be discovered, it is likely that if any flaws are found they will likely be either subtle vulnerabilities resulting from the parameters *within* the design or like X9.31, outright design flaws. Simon and Speck are unique in that they represent the only known, potential PSYCHO rejected at standardization time.

For an attacker, Dual EC is the gold-standard PSYCHO: a passive 'NOBUS' backdoor reliant on the combination of a backdoored design and parameters. Further, it is the PRG for which there is the strongest evidence of an actual PSYCHO attack. A potential small subgroup attack using RFC 5114 fits in the same taxonomical category, with adversarial introduction of weak parameters into the standard. However, as Dual EC had low market penetration during its lifetime, the actual impact of this PSYCHO may have been somewhat muted.

Affected	Suspected PSYCHO	Revoked/Rejected	Years Active	Flaw Type
X9.31	✗	✗	1985-2016	D
Dual_EC	✓	✓	2004-2016	D , P
CTR_DRBG	✗	✗	2006-	D , T
RFC 5114	✓	✗	2008-	P
Simon & Speck	✓	✓	—	D
Logjam	?	✗	1990s-2010s	R , P , C
TLS Attacks	?	updated	1990s-	R , C

Table 13: Comparison of Possible PSYCHOs. Flaw types are as follows: D corresponds to a design flaw (narrowly construed), P corresponds to poor parameter choices, C corresponds to intentionally introduced complexity, R corresponds to regulatory impositions, and T corresponds to inadequate threat model. Question marks indicate a lack of clarity or consensus as to whether foul-play was in any part responsible for the corresponding flaw. TLS is marked as updated as the IETF (and implementations) standardized mitigations against the relevant attacks. The revoked or rejected column indicates action only on the part of the relevant standardization body.

CTR_DRBG's place in the PSYCHO ecosystem is more subtle. While the specific attack I discovered operates in relatively constrained circumstances, the type of flaw on which it relies is a viable PSYCHO. By neglecting to include the whole category of side channel attacks within SP 800-90A, the PRGs were insufficiently assessed against such adversaries. A PSYCHO attacker could similarly introduce such flaws by working to *ensure* the exclusion of classes of attackers within a standard.

An interesting distinction can be drawn between Dual EC and the Logjam/Small Subgroup attacks. Despite the fact that all three rely on corrupted parameters, an adversary leveraging the latter two attacks is reliant on an advantage in available computational resources. All three are in some weak sense 'NOBUS', but that advantage is lost to other parties with similar resources and is further diminished as the cost of computing power decreases over time.

The Logjam, DROWN, FREAK, and POODLE TLS attacks exploited flaws either in the highly complex TLS state machine or in its implementation. Combined with regulatory imposition of compatibility for weaker parameters, this type of PSYCHO leaves significant room for plausible deniability. Although the protocol could have been made simpler (and thus more secure) without the agility and fragility required to support EXPORT cipher suites, the failure to securely integrate the weaker suites is difficult to pin

[216] Luke Valenta. *Measuring and Securing Cryptographic Deployments.* ProQuest Dissertations. 2019

on any individual or entity. While the cryptographic community is now better aware of the dangers of emphasizing cryptographic agility over simplicity [216], there are plausible rationales for introducing complexity, ripe for exploitation as a PSYCHO.

7.6 A Better Tomorrow

This section presents a series of recommendations and forward looking analyses to aid in the further development of secure systems.

7.6.1 NIST Standardization

While the National Institute of Standards and Technology (NIST) has historically pursued a number of routes to ensure the security of devices in government use, concern with the efficacy of third-party compliance audits, unclear standards and potential attempts to compromise cryptographic standards has caused a disconnect between compliance with standards and real world security. In conversations with the author, NIST officials recognized the above issues and committed that they were engaging in multiple avenues to improve standardization and compliance procedures. NIST highlighted the difficulty of effectively deprecating algorithms and cryptosystems, which typically have long lifespans. This contributes to ongoing concerns with historical, vulnerable, PRGs.

Standardization Failure. The failure of the NIST and Federal Information Processing Standard (FIPS) standardization process to protect against the long-known vulnerabilities in both X9.31 and CTR_DRBG, is surprising. The observation that the seed key must remain secret in the X9.17/X9.31 design was first noted decades ago, and yet none of the descriptions of the algorithm I could find mentioned the importance of generating an unpredictable key. Likewise, the flaws CTR_DRBG were highlighted from early in the SP 800-90 standardization process. While it's less clear how the issues in HMAC_DRBG and CTR_DRBG endured, the overall picture is one of a failed standardization process.

Certification Failure Security policy documents that indicated vulnerabilities should have been detected by the testing labs; the fact that they were not illustrates systemic issues with lab-based validation.

John Kelsey, (NIST employee and the original discoverer of the X9.31 flaw [8]), told us in personal communication that conflict of interest issues (the testing labs are paid by the vendors), as well as lab personnel skill, are on-going concerns with the efficacy of testing labs. To address these issues, NIST is transitioning to an automated validation program [217].

Eliminating Obsolete Standards. Kelsey related in personal communication that removing X9.17 key generation and the FIPS 186 PRGs from the standards that ultimately became NIST SP 800-90A was one of the first things he did when joining NIST. He was surprised to learn that in 2016 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; I 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.

Future of FIPS. FIPS 140-3 is expected to contain requirements for side-channel mitigations from the inclusion of NIST SP 800-140F, which has yet to be issued and becomes effective in September 2019 [218]. However, FIPS 140-2 Cryptographic Module Validation Program (CMVP) certifications will be continue to be issued at least through 2021. This is a promising step towards widespread deployment of side-channel-resistant cryptography; however, it remains to be seen how improved requirements for certifying modules will feed back into the design and standardization of more secure primitives.

[217] NIST. *Automated Cryptographic Validation Testing*. 2016

[218] National Institute of Standards and Technology. *The Federal Information Processing Standard (FIPS) Publication 140-3: DRAFT Security Requirements for Cryptographic Modules*. 2019

7.6.2 Broader Cryptographic Standards and Protocols

The purpose of standards, validation procedures and certification, is to help remove the onus from application developers to know and understand the details of the cryptographic attacks. Cryptographic recommendations from standards committees are often too weak or vague. Standards often exhibit 'fragility' whereby minimal departures from these recommendations quickly leads to the collapse of the security guarantees they purport to provide.

As a result, developers often forget or ignore edge cases, leaving many implementations at best, too close to vulnerable for comfort, and at worst, outright exploitable. Rather than provide the bare minimums for security, recommendations from standards bodies should be designed for defense-in-depth: a single mistake on the part of a developer does not have disastrous consequences for security. The principle of defense-in-depth has been a staple of the systems security community; cryptographic standards should similarly be designed to avoid fragility.

Protocol Design The interactions between cryptographic primitives and the needs of protocol designs can be complex. The after-the-fact introduction of RFC 5114 primes illustrates some of the unexpected difficulties: both IKE and SSH specified group validation only for safe primes, and a further RFC specifying extra group validation checks needed to be defined for IKE. Designing protocols that include unnecessary functionality, options, or extensions leaves room for implementation errors and burdens the task of formal security analysis. As illustrated in Chapter 3 and Chapter 4, IKE is a notorious example of a difficult-to-implement protocol with many edge cases. Just Fast Keying (JFK), a protocol created as a successor to IKEv1, was designed to be an exceedingly simple key exchange protocol without the unnecessarily complicated negotiations present in IKE [219]. However, the IETF instead standardized IKEv2, which is nearly as complicated as IKEv1. Protocols and cryptosystems should be designed with the developer in mind—easy to implement and verify, with limited edge cases. The worst possible outcome is a system that appears to work, but provides less security than expected.

To construct such cryptosystems, secure-by-default primitives are key. As shown in by my co-authors and I in "Measuring small subgroup attacks against Diffie–Hellman" [197], finite-field based Diffie–Hellman

[219] William Aiello et al. *Just fast keying: Key agreement in a hostile internet*. 2004

has many edge cases that make its correct use difficult, and which occasionally arise as bugs at the protocol level. For example, SSH and TLS allow the server to generate arbitrary group parameters and send them to the client, but provide no mechanism for the server to specify the group order so that the client can validate the parameters.

7.6.3 Policy Considerations

The potential uses of PSYCHOs are tempting tool for cryptanalytic agencies. Despite their potential efficacy, in this subsection I aim to provide a brief argument against their use.

Trusting the NSA. The NSA, like many countries' cryptanalytic agencies has a dual mission that encompasses both Signals Intelligence (SIGINT) and Information Assurance (IA). The IA mission is primarily defensive, focusing on how to protect and safeguard the information assets of both the United States government, and its businesses and citizenry, and political institutions. Conversely, the SIGINT mission relies on offensive capabilities. Natural tensions resulting from this dual mission impact the ability of cryptographers to take representations from the NSA at face value.

Though the NSA has always had motive to conceal otherwise unknown flaws, introduce new ones, and exploit whatever they can, the agency has at times acted publicly to secure cryptographic standards. The most notable instance of such public intervention being the agency's suggested fixes to the Data Encryption Standard (DES) SBoxes. While the agency did not contemporaneously provide a reason for their suggested fixes, it later became clear that the fixes hardened DES against differential cryptanalysis, which had yet to be discovered by the outside cryptographic community. As a result, the NSA attained a certain level of trust; for while cryptographers knew they had an adversary, when the NSA claimed to be aiding in the development of standards, they were given a certain amount of deference.

Unfortunately, this trust was compromised during the first 'Crypto War' of the 1990s, during which the NSA advocated for a key-escrow system to "allow Federal, State, and local law enforcement officials the ability to decode intercepted voice and data transmissions."[220]

[220] Glenn J McLoughlin. *The Clipper Chip A Fact Sheet Update*. Congressional Research Service, Sept. 1995

Respect for the agency was further corroded by the Snowden revelations, which to many indicated evidence of constitutionally-suspect intelligence gathering [221,222].

The NSA serves a praiseworthy mission of defending the U.S. from foreign threats, through its cryptanalytic expertise. However, its ability to do so may be hampered by current tensions with the academic and industrial cryptographic communities. The NSA would be well served to learn from both the blowback from the two crypto wars, and the harms illustrated herein. The agency should therefore refrain from engaging in PSYCHOs and work to rebuild the trust of its potential partners.

The NSA can also no longer rely on its institutional stature as justification for its actions within standards committees. While this may have been sufficient in an older era, cryptographic best practices have advanced and if the agency wishes to successfully interact with the community, it must adopt its new norms. Most importantly these includes transparency around design decisions and legitimate, good faith, engagement with the academic cryptographers and standardization bodies.

Historical NSA decryption capabilities. Classified NSA documents leaked by Edward Snowden and published by Der Spiegel [16] suggest that the NSA has passive decryption capabilities against some fraction of Internet Protocol Security (IPsec), Transport Layer Security (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 [93] (Boudot [223] 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 Deterministic Random Bit Generator (DRBG), and software exploits and malware ("implants"). These capabilities may have been supported by PSYCHOs conducted under "BULLRUN" and other monikers, as suggested by various leaked slides.

The reality is a likely combination of these techniques customized to vendors' vulnerabilities. While short of future declassification, the truth of the matter is unlikely to be known, these leaks provide at minimum circumstantial evidence of such activities.

[221] John Leyden. *Feds BANNED from DEF CON by founder (who is Obama's cyber-expert)*. The Register, 2013

[222] Spencer Ackerman. *NSA keeps low profile at hacker conventions despite past appearances*. The Guardian, 2014

[223] Fabrice Boudot. *On Improving Integer Factorization and Discrete Logarithm Computation using Partial Triangulation*. Cryptology ePrint Archive, Report 2017/758. 2017

While a PSYCHO compromising 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 of which I am are aware are the RSA BSAFE library, and Juniper ScreenOS.) By contrast, the X9.17/X9.31 PRG has been ubiquitous for decades.

Going Dark. In the recent debate over whether law enforcement is "going dark," some have proposed a mandate for cryptographic systems to allow "exceptional access": mechanisms for law enforcement to recover plaintext on demand. Dual EC represents a particularly promising template for such an exceptional access system. Either no one knows the value of d corresponding to NSA's point Q or only NSA does (barring a leak). In NSA jargon, a Dual EC-style backdoor would be "NOBUS": No one but the U.S. could exploit it.

The unauthorized change to ScreenOS's Dual EC constants made in 2012 illustrates a new threat: the ability for another party to apparently subvert a NOBUS back door for its own purposes, with only minimally detectable changes. Since the Federal Government is a NetScreen customer [224], the changes made in 2008 to add Dual EC to ScreenOS and to expose its output in IKE handshakes may have had negative repercussions for U.S. security, NOBUS or not.

Indeed, despite suspicion, there has been no incontrovertible evidence of Dual EC being used for kleptography. ScreenOS after the 2012 unauthorized change represents the deployment of Dual EC for which there is the strongest such evidence — and the party that took advantage of it may not have been NSA.

The U.S. government should think twice before employing PSYCHOs, for as this work evidences, the patient can rarely be confined to the asylum: surreptitiously introduced weaknesses will be discovered, and not always by a friendly doctoral candidate.

[224] Office of Personnel Management. *Juniper Network Firewall Maintenance Renewal.* FedBizOps.gov solicitation number M-13-00031. Online: https://www.fbo.gov/index?id=b3246ffee0a3e9c0ced948b3a8ebca7b. Sept. 2013

APPENDIX A

ETHICS & DISCLOSURE

The following ethics statements apply both to my contributions and the work performed by my co-authors across the various contributing publications.

Leges sine Moribus vanae.

Horace

A.1 Scanning

In Chapters 3–4 we perform whole Internet scans to assess the vulnerability of live hosts. While in Chapter 3 we demonstrate key recovery and decryption against hosts we do not own, the traffic we decrypt in our proof-of-concept is a handshake we initiate. In both instances 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.

[225] Fortinet. *FG-IR-17-245: DUHK Attack against Fortinet Products*. Online: https://fortiguard.com/psirt/FG-IR-17-245

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

A.2 X9.31

We disclosed the X9.31 and privilege escalation vulnerabilities identified in Chapter 3 to Fortinet in October 2016. Fortinet responded by releasing a patch for affected versions of FortiOS [225,226].

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 3 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.

A.3 CTR_DRBG

In 2019 we disclosed the vulnerabilities in CTR_DRBG we discuss in Chapter 5 to the security teams of OpenSSL, Fortinet, and NetBSD in May 2019. OpenSSL responded that these attacks are outside their threat model. Both NetBSD and Fortinet have since shared advisories and remediating measures to their customers. The Deterministic Random Bit Generator (DRBG) flaw in FortiOS was assigned CVE-2019-15703.

A.4 NIST

National Institute of Standards and Technology (NIST) decertified ANSI X9.31 pseudorandom number generator (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 the findings in Chapter 3, NIST noted several issues with cryptographic validation that they are planning to improve; we discuss these in detail in Section 7.6. With respect to the CTR_DRBG vulnerability, representatives from NIST indicated that the threat was

sufficiently attenuated so as not to necessitate a re-evaluation of the inclusion of CTR_DRBG within FIPS standards.

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 [217].

BIBLIOGRAPHY

[1] Jane Austen. *Pride and Prejudice*. 1813. (see p. 1)

[2] George W. Brown. *History of RAND's Random Digits*. Tech. rep. P-113. 1949 (see p. 2)

[3] John M. *The Difference Between RDRAND and RDSEED*. Online: https://software.intel.com/en-us/blogs/2012/11/17/the-difference-between-rdrand-and-rdseed. Intel Corporation, 2012 (see p. 2)

[4] Advanced Micro Devices (AMD). *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Tech. rep. 2015 (see p. 2)

[5] Elaine Barker and John Kelsey. *NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology, 2007. 109. (see pp. 2, 23, 52, 80, 83, 130)

[6] John Kelsey, Bruce Schneier, and Neils Ferguson. "Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator". In: *Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography*. SAC. London, UK, UK: Springer-Verlag, 1999. 13–33. (see p. 2)

[7] Stephan Müller. *Linux Random Number Generator — A New Approach*. Online: http://www.chronox.de/lrng/doc/lrng.html (see p. 2)

[8] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. "Cryptanalytic attacks on pseudorandom number generators". In: *Proceedings of the 5th International Workshop on Fast Software Encryption*. FSE. London, UK, UK: Springer-Verlag, 1998. 168–188. (see pp. 2, 7, 16, 21, 22, 25, 152)

[9] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, Daniel Wichs, Damien Vergnaud, and Daniel Wichs. "Security analysis of pseudo-random number generators with input: /dev/random is not robust". In: *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM. 2013. 647–658. (see pp. 2, 4, 10, 11, 16, 83, 129, 130)

[10] Falko Strenzke. "An analysis of OpenSSL's random number generator". In: *Proceedings, Part I, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9665*. Berlin, Heidelberg: Springer-Verlag, 2016. 644–669. (see p. 2)

[11] Sylvain Ruhault. SoK: Security Models for Pseudo-Random Number Generators. *Fast Software Encryption*, 506–544, 2017. (see pp. 2, 94)

[12] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. "How to Eat Your Entropy and Have It Too – Optimal Recovery Strategies for Compromised RNGs". In: *Proceedings of the 34th Annual International Cryptology Conference on Advances in Cryptology*. 2014. (see pp. 2, 16, 83, 129)

[13] Sandro Coretti, Yevgeniy Dodis, Harish Karthikeyan, and Stefano Tessaro. "Seedless Fruit Is the Sweetest: Random Number Generation, Revisited". In: *Proceedings of the 39th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. 2019. 205–234. (see pp. 2, 129)

[14] U.S. Congress. *Defending Secrets, Sharing Data: New Locks and Keys for Electronic Information: Appendix C*. Tech. rep. OTA-CIT-310. Office of Technology Assessment, Oct. 1987 (see p. 3)

[15] Aaron Margosis. *Why We're Not Recommending "FIPS Mode" Anymore*. Online: https://blogs.technet.microsoft.com/secguide/2014/04/07/why-were-not-recommending-fips-mode-anymore/. Microsoft, 2014 (see p. 3)

[16] National Security Agency. *Intro to the VPN Exploitation Process*. Media leak. Sept. 2010 (see pp. 4, 135, 155)

[17] Nicole Perlroth. *Government Announces Steps to Restore Confidence on Encryption Standards*. 2013 (see pp. 4, 15)

[18] Joanne Woodage and Dan Shumow. "An Analysis of the NIST SP 800-90A Standard." In: *38th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 2019. (see pp. 4, 10, 11, 83, 86, 92, 94, 95, 127, 130, 131)

[19] Niels Ferguson and Dan Shumow. "On the Possibility of a Back Door in the NIST SP800-90 Dual EC PRNG". In: *39th Annual International Cryptology Conference on Advances in Cryptology, Rump Session Talks*. Vol. 7. CRYPTO. 2007. (see pp. 7, 15, 19, 51, 53, 54, 80)

[20] Mihir Bellare, Kenneth G Paterson, and Phillip Rogaway. "Security of Symmetric Encryption against Mass Surveillance". In: *Proceedings of the 34th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. (see pp. 7, 16, 19, 128)

[21] Stephen Checkoway, Matthew Fredrikson, Wisconsin Madison, and Ruben Niederhagen. "On the Practical Exploitability of Dual EC in TLS Implementations". In: *Proceedings of the 23rd USENIX Security Symposium*. 2014. 319–335. (see pp. 7, 19, 51, 54, 63, 80, 106, 135)

[22] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. "A Formal Treatment of Backdoored Pseudorandom Generators". In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. 101–126. (see pp. 7, 16, 19, 83, 129)

[23] Jean Paul Degabriele, Kenneth G. Paterson, Jacob C.N. Schuldt, and Joanne Woodage. "Backdoors in pseudorandom number generators: Possibility and impossibility results". In: *Proceedings of the 36th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. Springer. 2016. 403–432. (see pp. 7, 17, 19)

[24] Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. "Practical State Recovery Attacks against Legacy RNG Implementations". In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. 2018. 265–280. (see p. 7)

[25] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. "A systematic analysis of the Juniper Dual EC incident". In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM. 2016. 468–479. (see pp. 8, 23, 135)

[26] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. "Pseudorandom Black Swans: Cache Attacks on CTR DRBG". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. 2020. (see p. 8)

[27] *15 U.S. Code §278g–3 - Computer standards program* (see p. 12)

[28] *44 U.S. Code §3541* (see p. 12)

[29] National Institute of Standards and Technology. *FIPS PUB 140-2 Security Requirements For Cryptographic Modules*. 2001 (see p. 12)

[30] Tammy Green, Carolyn French, Ashit Vora, and Ian Hall. "FIPS 140 Validation Process: Overview and Case Study". In: *International Cryptographic Module Conference*. 2019. (see p. 14)

[31] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication 800-131A*, 2011. (see p. 14)

[32] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. "Analysis of the linux random number generator". In: *Proceedings of the IEEE Symposium on Security and Privacy*. Vol. 2006. IEEE S&P. IEEE Press. 2006. 371–385. (see pp. 16, 130)

[33] Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau. *The Linux Pseudorandom Number Generator Revisited*. Cryptology ePrint Archive, Report 2012/251. 2012 (see p. 16)

[34] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the random number generator of the Windows operating system. *ACM Transactions on Information and System Security*, **13**: 10, 2009. (see p. 16)

[35] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. "Randomly failed! The state of randomness in current Java implementations". In: *Proceedings of the 13th International Conference on Topics in Cryptology*. CT-RSA'13. San Francisco, CA: Springer-Verlag, 2013. 129–144. (see p. 16)

[36] Matthew Green. *Random number generation: An illustrated primer*. Online: https://blog.cryptographyengineering.com/2012/02/21/random-number-generation-illustrated/. Aug. 2016 (see p. 16)

[37] Adam Young and Moti Yung. "Kleptography: Using cryptography against cryptography". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1997. 62–74. (see p. 16)

[38] Adam Young and Moti Yung. "The prevalence of kleptographic attacks on discrete-log based cryptosystems". In: *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. Springer. 1997. 264–276. (see p. 16)

[39] Ari Juels and Jorge Guajardo. "RSA key generation with verifiable randomness". In: *International Workshop on Public Key Cryptography*. Springer. 2002. 357–374. (see p. 16)

[40] Eu-Jin Goh, Dan Boneh, Benny Pinkas, and Philippe Golle. "The design and implementation of protocol-based hidden key recovery". In: *International Conference on Information Security*. Springer. 2003. 165–179. (see p. 16)

[41] Marc Stevens. "Counter-Cryptanalysis". In: *Proceedings of the 33rd Annual International Cryptology Conference on Advances in Cryptology*. Ed. by Canetti Ran and Juan A Garay. Vol. 8042. CRYPTO. Springer-Verlag, Aug. 2013. 129–146. (see p. 17)

[42] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. "When Private Keys are Public: Results From the 2008 Debian OpenSSL Vulnerability". In: *Proceedings of IMC 2009*. Ed. by Anja Feldmann and Laurent Mathy. ACM Press, Nov. 2009. 15–27. (see pp. 17, 23)

[43] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal*, **21**: 66–70, 1996. (see p. 17)

[44] bushing, marcan, segher, and sven. "PS3 Epic Fail". In: *27th Chaos Communication Congress*. 2010. (see p. 17)

[45] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices". In: *Proceedings of the 22nd USENIX Security Symposium*. 2012. (see pp. 17, 23, 32, 46)

[46] Arjen Lenstra, James P Hughes, Maxime Augier, Joppe Willem Bos, Thorsten Kleinjung, and Christophe Wachter. "Ron was wrong, Whit is right". In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology*. CRYPTO. 2012. 626–642. (see p. 17)

[47] Marcella Hastings, Joshua Fried, and Nadia Heninger. "Weak keys remain widespread in network devices". In: *Proceedings of the 2016 Internet Measurement Conference*. ACM. 2016. 49–63. (see pp. 18, 46)

[48] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. "Predictability of Android OpenSSL's Pseudo Random Number Generator". In: *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. Berlin, Germany: ACM, 2013. 659–668. (see p. 18)

[49] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz. "Reverse-Engineering a Cryptographic RFID Tag." In: *Proceedings of the 18th USENIX Security Symposium*. Vol. 28. 2008. (see p. 18)

[50] Daniel J. Bernstein, Yun An Chang, Chen Mou Cheng, Li Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. "Factoring RSA keys from certified smart cards: Coppersmith in the wild". In: *Advances in Cryptology - ASIACRYPT 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. 341–360. (see p. 18)

[51] George Kadianakis, Claudia V Roberts, Laura M Roberts, and Philipp Winter. *"Major key alert!" Anomalous keys in Tor relays*. Tech. rep. (see p. 18)

[52] Thomas Ristenpart and Scott Yilek. "When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography". In: *The Network and Distributed System Security Symposium (NDSS)*. 2010. (see p. 18)

[53] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael Swift. "Not-So-Random numbers in virtualized linux and the Whirlwind RNG". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. 2014. 559–574. (see p. 18)

[54] Alex Klyubin. *Some SecureRandom Thoughts*. Online: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html. Aug. 2013 (see p. 18)

[55] Accredited Standards Committee (ASC) X9, Financial Services. *ANSI X9.17: Financial Institution Key Management (Wholesale)*. Tech. rep. 1985 (see p. 21)

[56] Sharon 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 (see pp. 21, 22, 24, 30)

[57] Lawrence E. Bassham III and Sharon S. Keller. *The Random Number Generator Validation System (RNGVS)*. National Institute of Standards and Technology. 2005 (see pp. 22, 37)

[58] Manuel Blum and Silvio Micali. How to Generate Cryptographically Strong Sequences of Pseudo-random Bits. *SIAM Journal on Computing*, **13**: 850–864, 1984. (see p. 23)

[59] D Harkins and D Carrel. *The Internet Key Exchange (IKE)*. RFC 2409. Internet Engineering Task Force, 1998 (see pp. 27, 62, 63, 67, 70)

[60] M Schneider D. Maughan M. Schertler and J Turner. *Internet Security Association and Key Management Protocol*. RFC 2408. Internet Engineering Task Force, 1998 (see p. 27)

[61] C. Kaufman. *Internet Key Exchange (IKEv2) Protocol*. RFC 4306. Internet Engineering Task Force, 2005 (see pp. 28, 62, 68–70)

[62] National Institute of Standards and Technology. *CMVP Historical Validation List*. Feb. 2017 (see pp. 30, 76)

[63] Computer Security Resource Center, National Institute of Standards and Technology. *Cryptographic Algorithm Validation Program - RNG Validation List*. Online: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/validation/validation-list/rng (see p. 30)

[64] INZERO SYSTEMS. *FIPS 140-2 SECURITY POLICY FOR: INZERO GATEWAY*. Online: www.kmip.me/www3.cryptsoft.com/fips140/unpdf/140sp1841-1.html (see p. 32)

[65] Juniper Networks, Inc. *FIPS 140-2 SECURITY POLICY - SSG 140*. Online: https://www.juniper.net/documentation/hardware/netscreen-certifications/Security_Policy_SSG-140_ScreenOS_6_2.pdf (see p. 32)

[66] IDC 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: https://www.idc.com/getdoc.jsp?containerId=prUS41490016 (see p. 35)

[67] MITRE Corporation. *CVE-2016-8492* (see p. 35)

[68] MITRE Corporation. *CVE-2016-1909* (see p. 35)

[69] Fortinet. *Brief Statement Regarding Issues Found with FortiOS*. Online: https://blog.fortinet.com/2016/01/12/brief-statement-regarding-issues-found-with-fortios. Jan. 2016 (see p. 35)

[70] Kaspersky Labs. *Equation Group Questions and Answers*. https://web.archive.org/web/20150217023145/https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf. 2015 (see p. 35)

[71] Wladimir J van der Laan. *TADAQUEOUS moments*. Online: http://laanwj.github.io/2016/09/01/tadaqueos.html. Sept. 2016 (see p. 36)

[72] Denys Vlasenko. *BusyBox: The Swiss Army Knife of Embedded Linux*. Online: https://busybox.net/about.html. 2008 (see p. 37)

[73] National Institute of Standards and Technology. *FIPS PUB 186-4: Digital Signature Standard (DSS)*. 2013 (see pp. 39, 104, 141)

[74] William Herlands, Thomas Hobson, and Paula Donovan. "Effective Entropy for Memory Randomization Defenses". In: *7th USENIX Workshop on Cyber Security Experimentation and Test*. Lincoln Labratory, Aug. 2014. (see p. 40)

[75] John Franks, Phillip M Hallam-Baker, Jeffery L Hostetler, Scott D Lawrence, Paul J Leach, Ari Luotonen, and Lawrence C Stewart. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. Internet Engineering Task Force, June 1999 (see p. 45)

[76] Dan Goodin. *Group claims to hack NSA-tied hackers, posts exploits as proof*. Aug. 2016 (see p. 45)

[77] Berry Schoenmakers and Andrey Sidorenko. *Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator*. Cryptology ePrint Archive, Report 2006/190. 2006 (see pp. 51, 71, 80)

[78] Kristian Gjøsteen. *Comments on Dual-EC-DRBG/NIST SP 800-90, Draft December 2005*. Mar. 2006 (see pp. 51, 71, 80, 127)

[79] James R. Clapper. *Worldwide Threat Assessment of the U.S. Intelligence Community*. Statement for the record, Senate Armed Services Committee. Online: http://www.armed-services.senate.gov/imo/media/doc/Clapper_02-09-16.pdf. Feb. 2016 (see p. 52)

[80] Juniper. *2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756)*. Online: https://kb.juniper.net/InfoCenter/index?page=content{&}id=JSA10713. Dec. 2015 (see pp. 52, 56, 58, 78)

[81] John Kelsey. *Dual EC in X9.82 and SP 800-90*. https://csrc.nist.gov/csrc/media/projects/crypto-standards-development-process/documents/dualec_in_x982_and_sp800-90.pdf. May 2014 (see pp. 52, 80)

[82] National Institute of Standards and Technology. *NIST Opens Draft Special Publication 800-90A, Recommendation For Random Number Generation Using Deterministic Random Bit Generators for Review and Comment*. Sept. 2013 (see pp. 54, 55, 78)

[83] Juniper Networks. *Juniper Networks Product Information about Dual_EC_DRBG*. Knowledge Base Article KB28205. Oct. 2013 (see pp. 55, 80)

[84] T Ylonen and C Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Internet Engineering Task Force, Jan. 2006 (see p. 56)

[85] H D Moore. *CVE-2015-7755: Juniper ScreenOS Authentication*. Dec. 2016 (see p. 56)

[86] Ralf-Philipp Weinmann. *Some Analysis of the Backdoored Backdoor*. Online: https://rpw.sh/blog/2015/12/21/the-backdoored-backdoor/. Dec. 2015 (see p. 57)

[87] Accredited Standards Committee (ASC) X9, Financial Services. *ANSI X9.31-1998: Digital Signatures using Reversible Algorithms for the Financial Services Industry (rDSA)*. 1998 (see p. 59)

[88] S Kent and K Seo. *Security Architecture for the Internet Protocol*. RFC 4301. Internet Engineering Task Force, Dec. 2005 (see p. 62)

[89] Juniper Networks. *Concepts & Examples ScreenOS Reference Guide: Virtual Private Networks*. Rev. 02, Dec. 2012. (see p. 66)

[90] StrongSwan. *strongSwan: the OpenSource IPsec-based VPN Solution*. Nov. 2015 (see p. 69)

[91] S Kent. *IP Encapsulating Security Payload (ESP)*. RFC 4303. Internet Engineering Task Force, Nov. 2005 (see p. 70)

[92] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. "ZMap: Fast Internet-wide Scanning and Its Security Applications". In: *Proceedings of the 22nd USENIX Security Symposium*. Ed. by Sam King. August. USENIX. 2013. 605–619. (see p. 73)

[93] 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. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. 2015. (see pp. 73, 135, 143, 155)

[94] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. "A Search Engine Backed by Internet-Wide Scanning". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM Press, Oct. 2015. 542–53. (see p. 74)

[95] Accredited Standards Committee (ASC) X9, Financial Services. *ANSI X9.82-3-2007: Random Number Generation, Part 3: Deterministic Random Bit Generators*. 2007 (see p. 78)

[96] Elaine Barker and John Kelsey. *NIST Special Publication 800-90A Revision 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Tech. rep. National Institute of Standards and Technology, June 2015 (see p. 80)

[97] S. Shane, J. Larson, and N. Pearlroth. *N.S.A. Able to Foil Basic Safeguards of Privacy on Web - The New York Times*. 2013 (see p. 80)

[98] Bob Worrall. *Advancing the Security of Juniper Products*. Online: http://forums.juniper.net/t5/Security-Incident-Response/Advancing-the-Security-of-Juniper-Products/ba-p/286383. Jan. 2016 (see p. 80)

[99] Mihir Bellare and Bennet Yee. "Forward-security in private-key cryptography". In: *Cryptographers' Track at the RSA Conference*. Springer. 2003. 1–18. (see p. 83)

[100] Boaz Barak and Shai Halevi. "A model and architecture for pseudo-random generation with applications to/dev/random". In: *Proceedings of the 12th ACM SIGSAC Conference on Computer and communications security*. ACM CCS. ACM. 2005. 203–212. (see p. 83)

[101] Mario Cornejo and Sylvain Ruhault. "Characterization of real-life PRNGs under partial state corruption". In: *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM. 2014. 1004–1015. (see p. 83)

[102] Daniel J Bernstein. *Cache-timing attacks on AES*. Online: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005 (see pp. 84, 86, 97)

[103] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and counter-measures: The case of AES". In: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*. CT-RSA. San Jose, CA: Springer-Verlag, 2006. 1–20. (see pp. 84–86, 88, 89, 97, 99, 101)

[104] C Percival. "Cache missing for fun and profit". In: *BSDCan 2005*. BSDCan, 2005. (see p. 84)

[105] Yuval Yarom and Katrina Falkner. "Flush + Reload : a High Resolution , Low Noise , L3 Cache Side-Channel Attack". In: *Proceedings of the 24th USENIX Security Symposium*. 2014. 1–14. (see pp. 84, 88, 89)

[106] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks : Automating Attacks on Inclusive Last-Level Caches This paper is included in the Proceedings of the Cache Template Attacks :" in: *Proceedings of the 25th USENIX Security Symposium*. 2015. 897–912. (see pp. 84, 88, 89, 97)

[107] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. 2018. (see pp. 84, 89)

[108] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *Proceedings of the 27th USENIX Security Symposium*. 2017. 991–1008. (see pp. 84, 89, 115)

[109] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. Tech. rep. 2018 (see pp. 84, 89)

[110] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel

Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *Proceedings of the 28th USENIX Security Symposium*. 2018. (see pp. 84, 89)

[111] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A systematic evaluation of transient execution attacks and defenses". In: *28th USENIX Security Symposium*. 2019. 249–266. (see p. 84)

[112] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *Proceedings of the International Symposium on Computer Architecture*. 2014. 361–372. (see p. 84)

[113] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. MAGIC: Malicious aging in circuits/cores. *ACM Transactions on Architecture and Code Optimization*, **12**: 5, 2015. (see p. 84)

[114] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F. Haratsch. "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017. (see p. 84)

[115] Anil Kurmus, Nikolas Ioannou, Matthias Neugschwandtner, Nikolaos Papandreou, and Thomas Parnell. "From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks". In: *11th USENIX Workshop on Offensive Technologies*. 2017. 1–9. (see p. 84)

[116] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. "RAMBleed: Reading Bits in Memory Without Accessing Them". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. 2020. (see p. 84)

[117] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-tenant side-channel attacks in PaaS clouds". In: *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM. 2014. 990–1003. (see p. 84)

[118] Joseph Bonneau. *Robust Final-Round Cache-Trace Attacks Against AES*. Cryptology ePrint Archive, Report 2006/374. 2006 (see pp. 85, 97, 99)

[119] Michael Neve and Jean Pierre Seifert. "Advances on access-driven cache attacks on AES". In: *Proceedings of the 13th International Conference on Selected Areas in Cryptography*. SAC. Montreal, Canada: Springer-Verlag, 2007. 147–162. (see pp. 85, 86, 97–99)

[120] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache games - Bringing access-based cache attacks on AES to practice". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. 2011. 490–505. (see pp. 86, 88, 97)

[121] Samira Briongos, Pedro Malagón, Juan Mariano de Goyeneche, and Jose M. Moya. Cache misses and the recovery of the full AES 256 Key. *Applied Sciences (Switzerland)*, **9**: 944, 2019. (see p. 86)

[122] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. "Are AES x86 cache timing attacks still feasible?" In: *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. 2012. 19–24. (see p. 86)

[123] Fan Zhang. *mbedtls-SGX*. Online: https://github.com/bl4ck5un/mbedtls-SGX. GitHub, 2018 (see pp. 86, 87, 122)

[124] Henric Jungheim. *NIST SP 800-90Ar1 CTR_DRBG*. Online: https://henric.org/random/#nistrng. 2019 (see pp. 86, 96)

[125] Shay Gueron and Yehuda Lindell. "GCM-SIV: Full nonce misuse-resistant authenticated encryption at under one cycle per byte". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. 2015. 109–119. (see p. 87)

[126] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a Minute! A fast, Cross-VM Attack on AES". In: *Research in Attacks, Intrusions and Defenses*. Ed. by Angelos Stavrou, Herbert Bos, and Georgios Portokalidis. Cham: Springer International Publishing, 2014. 299–319. (see pp. 88, 89, 97)

[127] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. [ARM TrustZone] TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *Cryptology ePrint Archive, Report 2016/980*, **2016**: 980, 2016. (see pp. 88, 97)

[128] Raphael Spreitzer and Thomas Plos. "Cache-access pattern attack on disaligned AES t-tables". In: *Proceedings of the 4th International Conference on Constructive Side-Channel Analysis and Secure Design*. COSADE'13. Paris, France: Springer-Verlag, 2013. 200–214. (see p. 88)

[129] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A fast and stealthy cache attack". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*. DIMVA 2016. San Sebastian, Spain: Springer-Verlag, 2016. 279–299. (see p. 88)

[130] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, **23**: 37–71, 2010. (see pp. 88, 89)

[131] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. "Prime + Abort : A Timer-Free High-Precision L3 Cache Attack using Intel TSX This paper is included in the Proceedings of the". In: *Proceedings of the 26th USENIX Security Symposium*. 2017. (see p. 88)

[132] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. "Scheduler-based Defenses against Cross-VM Side-channels". In: *UProceedings of the 24th USENIX Security Symposium*. 2014. 687–702. (see pp. 88, 89)

[133] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, Gauss, and Reload: A cache attack on the BLISS lattice-based signature scheme". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016. 323–345. (see pp. 88, 89)

[134] Rishabh Poddar, Amit Datta, and Chester Rebeiro. "A cache trace attack on CAMELLIA". In: *Security Aspects in Information Technology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. 144–156. (see p. 88)

[135] The OpenSSL Project. *OpenSSL: The Open Source toolkit for SSL/TLS*. http://openssl.org/. 2003 (see p. 89)

[136] GnuPG Project. *GnuPG*. Online: www.gnupg.org. 2019 (see p. 89)

[137] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *Proceedings of the 26th USENIX Security Symposium*. 2016. 557–574. (see p. 89)

[138] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies*. 2017. (see pp. 89, 115)

[139] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX amplifies the power of cache attacks". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. SysTEX'17. Shanghai, China: ACM, 2017. 4:1–4:6. (see pp. 89, 115)

[140] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *Proceedings of the IEEE Symposium on Security and Privacy*. Vol. 2015-July. IEEE S&P. IEEE. 2015. 640–656. (see pp. 89, 115, 120)

[141] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. "Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves". In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. 2017. (see pp. 89, 122)

[142] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. "To BLISS-B or not to be". In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. 2017. 1843–1855. (see p. 89)

[143] Cesar Garcia and Billy Brumley. "Constant-Time Callees with Variable-Time Callers". In: *Proceedings of the 26th USENIX Security Symposium*. 2017. (see p. 89)

[144] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. *CoRR*, 2018. (see p. 89)

[145] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-Flight Data Load". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. 2019. (see p. 89)

[146] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. *Fallout: Reading Kernel Writes From User Space*. 2019 (see p. 89)

[147] Matthew J Campagna. *Security Bounds for the NIST Codebook-based Deterministic Random Bit Generator.* 2006 (see p. 94)

[148] Wilson Kan. *Analysis of Underlying Assumptions in NIST DRBGs*. 2007 (see p. 94)

[149] Shoichi Hirose. "Security analysis of DRBG using HMAC in NIST SP 800-90". In: *WISA 2008: Information Security Applications*. Springer. 2008. 278–291. (see p. 94)

[150] Thomas Shrimpton and R. Seth Terashima. "Salvaging weak security bounds for blockcipher-based constructions". In: *ASIACRYPT*. 2016. (see p. 94)

[151] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. "Verified Correctness and Security of mbedTLS

HMAC-DRBG". In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM. 2017. 2007–2020. (see p. 94)

[152] Daniel J Bernstein. *Fast-key-erasure random-number generators*. Online: https://blog.cr.yp.to/20170723-random.html. July 2017 (see p. 94)

[153] Matthew D Green. *Twitter Thread on OpenSSL*. Online: https:/twitter.com/matthew_d_green/status/1115013260783255558 (see p. 96)

[154] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van De Pol, and Yuval Yarom. "Amplifying side channels through performance degradation". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM. 2016. 422–435. (see p. 101)

[155] FX Breu, S Guggenbichler, and JC Wollmann. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313. Internet Engineering Task Force, 2008, 1–19 (see p. 103)

[156] Matthew D Green. *Wonk post: chosen ciphertext security in public-key encryption (Part 2)*. Online: https://https://blog.cryptographyengineering.com/2018/07/20/wonk-post-chosen-ciphertext-security-in-public-key-encryption-part-2/. 2018 (see p. 103)

[157] Daniel Bleichenbacher. "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1". In: *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. London, UK, UK: Springer-Verlag, 1998. 1–12. (see p. 103)

[158] Bellare Mihir and Phillip Rogaway. *PSS: Provably Secure Encoding Method for Digital Signatures*. Submission to IEEE P1363. 1998 (see p. 104)

[159] E Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force, 2018 (see pp. 104, 106)

[160] B. Kaliski, J. Jonsson, and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Internet Engineering Task Force, 2016 (see p. 104)

[161] Matthew D Green. *The strange story of "Extended Random"*. Online: http://blog.cryptographyengineering.com/2017/12/19/the-strange-story-of-extended-random/. 2017 (see p. 106)

[162] [openssl-users] mailing list. *Openssl software failure for RSA 16K modulus*. Online: https://mta.openssl.org/pipermail/openssl-users/2016-July/004056.html. 2016 (see p. 109)

[163] [openssl-dev] mailing list. *[openssl.org #4063] Re: Client Hello longer than 2^{14} bytes are rejected*. Online: https://mta.openssl.org/pipermail/openssl-dev/2015-September/002860.html. 2015 (see p. 109)

[164] Jean-Philippe Aumasson and Antony Vennard. *Audit of OpenSSL's randomness generation*. Online: https://ostif.org/wp-content/uploads/2018/09/opensslrng-audit-report.pdf. 2018 (see p. 111)

[165] Quarkslab SAS. *OpenSSL Security Assessment*. Online: https://ostif.org/wp-content/uploads/2019/01/18-04-720-REP_v1.2.pdf. 2019 (see p. 111)

[166] OpenSSL. *SSL/TLS Client*. 2018 (see p. 112)

[167] Mathy Vanhoef and Eyal Ronen. "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd". In: *Proceedings of the IEEE Symposium on Security and Privacy)*. IEEE S&P. IEEE. 2020. (see p. 114)

[168] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM. 2017. 4. (see pp. 114, 115)

[169] Marcus Hähnel, Weidong Cui, Marcus Peinado, and Tu Dresden. "High-Resolution Side Channels for Untrusted Operating Systems". In: *USENIX Annual Technical Conference*. 2017. 299–312. (see pp. 114, 115)

[170] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. ACM CCS. ACM. 2017. 2421–2434. (see p. 114)

[171] Matthew H. *Intel SGX for Dummies (Intel SGX) Design Objectives*. Online: software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx. 2013 (see p. 115)

[172] Alex Biryukov. In: *Substitution–Permutation (SP) Network*. Henk C. A. van Tilborg and Sushil Jajodia, eds. 1268–1268. Boston, MA: Springer US, 2011. (see p. 115)

[173] Eli Biham and Adi Shamir. "Differential fault analysis of secret key cryptosystems". In: *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. London, UK, UK: Springer-Verlag, 1997. 513–525. (see p. 116)

[174] Thomas Roche, Victor Lomné, and Karim Khalfallah. "Combined fault and side-channel attack on protected implementations of AES". In: *Proceedings of the 10th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*. CARDIS'11. Leuven, Belgium: Springer-Verlag, 2011. 65–83. (see p. 121)

[175] Josh Jaffe. "A First-Order DPA Attack Against AES in Counter Mode with Unknown Initial Counter". In: *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES. Vienna, Austria: Springer-Verlag, 2007. 1–13. (see p. 121)

[176] Eyal Ronen, Adi Shamir, Achi Or Weingarten, and Colin Oflynn. "IoT Goes Nuclear: Creating a Zigbee Chain Reaction". In: *Proceedings of the IEEE Symposium on Security and Privacy*. Vol. 16. IEEE S&P 1. 2018. 54–62. (see p. 122)

[177] Intel. *Intel Software Guard Extensions SSL*. Online: https://github.com/intel/intel-sgx-ssl. 2017 (see pp. 124, 129)

[178] Joachim Breitner and Nadia Heninger. "Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies". In: *Financial Cryptography*. 2019. (see p. 124)

[179] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *Cryptology ePrint Archive 2014/140.pdf* , **2014**: 12–14, 2014. (see p. 124)

[180] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. Internet Engineering Task Force, Aug. 2013 (see p. 124)

[181] Adam Young and Moti Yung. "Kleptography: Using cryptography against cryptography". In: *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'97. Konstanz, Germany: Springer-Verlag, 1997. 62–74. (see p. 126)

[182] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. "Hedged Public-Key Encryption: How to Protect Against Bad Randomness". In: *Proceedings of Asiacrypt 2009*. Ed. by Mitsuru Matsui. Vol. 5912. LNCS. Springer-Verlag, Dec. 2009. 232–49. (see p. 128)

[183] M. Fischlin, C. Janson, and S. Mazaheri. "Backdoored Hash Functions: Immunizing HMAC and HKDF". In: *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*. 2018. 105–118. (see p. 129)

[184] Wtdrog. *Systemd Issue #11810 - Can't suspend again after suspending one time*. May 2019 (see p. 130)

[185] Hin-Tak Leung. *Redhat Bug 1150286 - rdrand instruction fails after resume on AMD CPU*. Online: https://bugzilla.redhat.com/show_bug.cgi?id=1150286. 2019 (see p. 130)

[186] Daniel J. Bernstein. *Entropy Attacks!* Online: http://blog.cr.yp.to/20140205-entropy.html. 2014 (see p. 130)

[187] Jake Edge. *Fixing getrandom()*. Online: https://lwn.net/Articles/800509/. LWN.net, 2019 (see p. 130)

[188] Steve Marquess. *Flaw in Dual EC DRBG (no, not that one)*. Online: http://marc.info/?l=openssl-announce&m=138747119822324&w=2. 2013 (see p. 131)

[189] Sean Gallagher. *Photos of an NSA "upgrade" factory show Cisco router getting implant*. 2014 (see p. 135)

[190] Jacob Appelbaum, Judith Horchert, and Christian Stöcker. Shopping for spy gear: Catalog advertises NSA toolbox. *Der Spiegel*, 2013. (see p. 135)

[191] Jacob L. Wright. Warfare and Wanton Destruction: A Reexamination of Deuteronomy 20: 19-20 in Relation to Ancient Siegecraft. *Journal of Biblical Literature*, **127**: 423–458, 2008. (see p. 136)

[192] Melinda Reed, John F Miller, and Paul Popick. *Supply chain attack patterns: Framework and Catalog*. Tech. rep. 2013 (see pp. 137, 138)

[193] Daniel DiMase, Zachary A. Collier, Kenneth Heffner, and Igor Linkov. Systems engineering framework for cyber physical security and resilience. *Environment Systems and Decisions*, **35**: 291–300, 2015. (see p. 138)

[194] Bhanu Sood, Diganta Das, and Michael Pecht. Screening for counterfeit electronic parts. *Journal of Materials Science: Materials in Electronics*, **22**: 1511, 2011. (see p. 138)

[195] Dan Kaufman. *An analytical framework for cyber security*. Tech. rep. Defense Advanced Research Projects Agency, 2011 (see p. 138)

[196] Chae Hoon Lim and Pil Joong Lee. "A key recovery attack on discrete log-based schemes using a prime order subgroup". In: *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO. 1997. (see p. 141)

[197] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J Alex Halderman, and Nadia Heninger. "Measuring small subgroup attacks against Diffie-Hellman." In: vol. 2016. 2016. 995. (see pp. 141, 153)

[198] Elaine B Barker, Don Johnson, and Miles E Smid. *SP 800-56A. recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)*. Tech. rep. 2007 (see p. 142)

[199] M Lepinski and Steve Kent. *Additional Diffie-Hellman Groups for Use with IETF Standards*. RFC 5114. Internet Engineering Task Force, 2008 (see p. 142)

[200] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the Elliptic curve Digital Signature Algorithm with partially known nonces. *Designs, codes and cryptography*, **30**: 201–217, 2003. (see p. 143)

[201] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. "DROWN: Breaking TLS with SSLv2". In: *Proceedings of the 25th USENIX Security Symposium*. August. 2016. 1–18. (see p. 145)

[202] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. *This POODLE bites: exploiting the SSL 3.0 fallback*. Tech. rep. 2014 (see p. 145)

[203] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE S&P. May 2015. 535–552. (see p. 145)

[204] Tomer Ashur and Atul Luykx. "An Account of the ISO/IEC Standardization of Simon and Speck". In: 2019. (see pp. 146, 148)

[205] National Security Agency. *Algorithms to Support the Evolution of Information Assurance Needs*. Tech. rep. (see p. 146)

[206] Tomer Ashur. *[PATCH v2 0/5] crypto: Speck support*. Linux Crypto, Mailing List. 2018 (see p. 147)

[207] Tomer Ashur and Atul Luykx. "How we avoided a second Dual EC DRBG fiasco (Simon and Speck will not be standardized as encryption algorithms in ISO)". In: *38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rump Session*. 2018. (see p. 147)

[208] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *Notes on the design and analysis of SIMON and SPECK*. 2017 (see p. 147)

[209] Ed Targett. *NSA Ciphers "Simon and Speck" Are Dead – But Not Entirely Buried Says ISO*. Online: https://www.cbronline.com/news/nsa-ciphers-iso. Computer Business Review, 2018 (see pp. 147, 148)

[210] Eric Biggers. *crypto: speck - add support for the Speck block cipher*. Online: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da7a0ab5b4babbe5d7a46f852582be06a00a28f0. 2018 (see p. 148)

[211] Michael Larabel. *EXT4/Fscrypt Changes For Linux 4.18: Speck File-System Encryption Being Added*. Online: https://www.phoronix.com/scan.php?page=news_item&px=EXT4-Fscrypt-Speck-Crypto-4.18. 2018 (see p. 148)

[212] jean-michel. *ALERT! Linux kernel 4.17 contains a backdoor.* Obarun Forum, Online: https://forum.obarun.org/viewtopic.php?id=584. 2018 (see p. 148)

[213] Abhishek Prakash. *NSA's Encryption Algorithm in Linux Kernel is Creating Unease in the Community*. 2018 (see p. 148)

[214] Jason A. Donenfeld. *crypto: speck - remove Speck*. Online: https://git.kernel.org/pub/scm/linux/kernel/git/herbert/cryptodev-2.6.git/commit/?anzwix=1&id=578bdaabd015b9b164842c3e8ace9802f38e7ecc. 2018 (see p. 148)

[215] Theodore T'so. *Re: [PATCH] crypto: remove speck*. Online: https://lkml.org/lkml/2018/8/7/30. 2018 (see p. 148)

[216] Luke Valenta. *Measuring and Securing Cryptographic Deployments*. ProQuest Dissertations. 2019 (see pp. 150, 151)

[217] NIST. *Automated Cryptographic Validation Testing*. 2016 (see pp. 152, 159)

[218] National Institute of Standards and Technology. *The Federal Information Processing Standard (FIPS) Publication 140-3: DRAFT Security Requirements for Cryptographic Modules*. 2019 (see p. 152)

[219] William Aiello, Steven Michael Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Angelos D Keromytis, and Omer Reingold. *Just fast keying: Key agreement in a hostile internet*. 2004 (see p. 153)

[220] Glenn J McLoughlin. *The Clipper Chip A Fact Sheet Update*. Congressional Research Service, Sept. 1995 (see p. 154)

[221] John Leyden. *Feds BANNED from DEF CON by founder (who is Obama's cyber-expert)*. The Register, 2013 (see p. 155)

[222] Spencer Ackerman. *NSA keeps low profile at hacker conventions despite past appearances*. The Guardian, 2014 (see p. 155)

[223] Fabrice Boudot. *On Improving Integer Factorization and Discrete Logarithm Computation using Partial Triangulation*. Cryptology ePrint Archive, Report 2017/758. 2017 (see p. 155)

[224] Office of Personnel Management. *Juniper Network Firewall Maintenance Renewal*. FedBizOps.gov solicitation number M-13-00031. Online: https://www.fbo.gov/index?id=b3246ffee0a3e9c0ced948b3a8ebca7b. Sept. 2013 (see p. 156)

[225] Fortinet. *FG-IR-17-245: DUHK Attack against Fortinet Products*. Online: https://fortiguard.com/psirt/FG-IR-17-245 (see pp. 157, 158)

[226] Fortinet. *FG-IR-16-067: FortiOS local privilege escalation via malicious use of USB storage devices*. Online: http://fortiguard.com/psirt/FG-IR-16-067 (see pp. 157, 158)

Warm copper rays streak the golden haze, as peace settles down on Penn



Sic itur ad astra...