UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BADRANDOM: THE EFFECT AND MITIGATIONS FOR LOW ENTROPY RANDOM NUMBERS IN TLS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

COMPUTER SCIENCE

by

James Prescott Hughes

December 2021

The Dissertation of James Prescott Hughes
is approved:

Table of Contents

List of Figures

List of Tables

Abstract

BadRandom: The effect and mitigations for low entropy random numbers in TLS

by

James Prescott Hughes

E-commerce has become critical to everyday life and how businesses and governments operate. The Internet's global reach is a significant factor in E-commerce success, but it ultimately would not be possible without secure communications. The IETF Transport Layer Security (TLS) protocol is used for almost all Internet traffic security, but TLS is not as secure as the general public believes it to be.

We know that the current TLS *protocol* is proven secure, but it is uncertain if the *implementations* live up to that promise. The history of random number generators that have not been as random as expected has led us to question the security of TLS.

Random numbers are the key to any cryptographic protocol's security. These numbers separate the attacker from the attacked. The proof assumes that all random numbers are perfectly random. If the actual implementations of TLS's random numbers are not perfectly random, the protocol's security proof is not applicable at best and worthless at worst.

We measured the randomness of actual TLS traffic to discover if TLS random numbers are indeed random. The TLS protocol has a raw random value the protocol uses to ensure that the connection is fresh. We captured two years of Internet traffic to and from UCSC to determine if the exposed raw random values are random.

The findings are disturbing. We found client implementations that do not offer any security because of simple programming mistakes. We found other insecure closed source client implementations and used them to demonstrate that the TLS protocol is fragile to insufficiently random numbers. One can not solely blame the programmers. We have discovered that the fragility of the TLS protocol contributes to these failures by allowing passive monitoring to identify these vulnerable implementations.

The IETF has standardized fragile protocols with at least the tacit approval of governments worldwide; we can do better. We propose a new proven secure TLS Authenticated Key Agreement Protocol that hides the implementation and is robust to random numbers that are less than perfect.

Acknowledgments

I want to thank Kathy, my children, and my friends who said in the beginning “Do it!” and in the end, said “Get it done!”. I wish to thank Professor Long and the committee; Ethan Miller, Arjen Lenstra, Whitfield Diffie and Christoph Schuba for their advice and motivation for my success. I wish to thank the UCSC Storage System Research Center (SSRC) Professors and students for their support. I wish to thank the UCSC IT Vice Chancellor Van Williams for his research support in giving me access to the University’s Zeek data, and Mike Ware for keeping the data flowing and helping out where he could.

No financial aid was received for this work. I am not an employee of UCSC, and I have not made use of any gift, grant, contract funds, or research facilities of UCSC in connection with my Ph.D. work at UCSC.

Glossary

AKAP The Authenticated Key Agreement Protocol (AKAP) is a Key Agreement Protocol (KAP) [82] that additionally authenticates the two parties and prevents Man In The Middle (MITM) attacks 53, 54, 67

backdoor Backdoors are software vulnerabilities that are intentionally planted in software to facilitate exploitation [63] 41

bag In mathematics, a bag (or multiset, or mset) is a modification of the concept of a set that, unlike a set, allows for multiple instances for each of its elements 15

bugdoor Bugdoors are a specific type of backdoor that disguise themselves as conveniently exploitable yet hard-to-spot bugs [63] 41

CDH Computational Diffie Hellman problem (CDH) is the problem of calculating a discrete log of a number. Specifically, given $(X, g) \in \mathbb{Z}_p$ determine the x such that $X = g^x \mod p$ 61, 65

CFRG The Crypto Forum Research Group (CFRG) is a general forum for discussing and reviewing uses of cryptographic mechanisms, both for network security in general and for the IETF in particular. The CFRG serves as a bridge between theory and practice [41] 11

computationally infeasible A problem that, in theory is computable, but in practice required more computing power than is possible 67

covert channel A channel is covert if it is neither designed nor intended to transfer information [77]. Covert channels are generally used to covertly exfiltrate data 51

CSPRNG Cryptographically Secure Pseudo Random Number Generator, an algorithm that takes a seed or other entropy and stretches it into a longer set of random numbers that are indistinguishable from actual random numbers 21, 26, 48, 49, 76

D-H Diffie Hellman (D-H) Key Agreement Protocol (KAP) allows Alice and Bob to agree on a key with Eve listening in such a way as to make the key that they agree is the same and that it is computationally infeasible for Eve to discover the agreed key. Given ephemeral random numbers $x \in \mathbb{Z}_p^*$ and $y \in \mathbb{Z}_p^*$ compute the ephemeral public keys $X = g^x$ and $Y = g^y$ with generator g in mod p. The agreed upon key is $k = g^{x \cdot y}$ which can be calculated by either $k = X^y$ or $k = Y^x$. The system is secure as long as there is no man in the middle attack (MITM) or x or y are compromised 56, 57

DDH Decisional Diffie Hellman assumption (DDH), impossible to differentiate between the two sets of numbers, (g^a, g^b, g^{ab}) and (g^c, g^d, g^e). It is important to note that the further assumption is that numbers (a, b, c, d, e) are random in $\mathbb{Z}_p$ and independent [72] 46, 48, 55, 66, 67

Ideal Random Sequence Each bit of an ideal random sequence is unpredictable and unbiased, with a value that is independent of the values of the other bits in the sequence. Prior to the observation of the sequence, the value of each bit is equally likely to be 0 or 1, and, the probability that a particular bit will have a particular value is unaffected by knowledge of the values of any or all of the other bits. An ideal random sequence of n bits contains n bits of entropy. [89] 76

IETF The mission of the IETF is to make the Internet work better by producing high quality, relevant technical documents that influence the way people design, use, and manage the Internet [5] xiii, 3, 5, 63

Internet traffic Internet traffic is the flow of data within the entire Internet, or in certain network links of its constituent networks. Common traffic measurements are total volume, in units of multiples of the byte, or as transmission rates in bytes per certain time units. [46] 12, 19, 50, 52

IoT The Internet of things describes devices that are not user centric, that is neither a human at a web browser or a web server. Generally sensors, actuators or other “things” 3

IRTF The Internet Research Task Force (IRTF) focuses on longer term research issues related to the Internet while the parallel organization, Internet Engineering Task Force (IeTF) [5], focuses on the shorter term issues of engineering and standards making [42] 11

KAP Key-agreement protocol (KAP) is a protocol whereby two or more parties can agree on a key in such a way that both influence the outcome [82] xi, 54, 56, 58, 62, 71

KEM Key encapsulation mechanisms (KEMs) are a class of encryption techniques designed to secure symmetric cryptographic key material for transmission using asymmetric (public-key) algorithms [74] 47, 71

kleptography Kleptography deals with employing and generating cryptographically secure covert channels [12] 4, 51

MITM The Man In The Middle is an attack against a conversation between two parties where the attacker actively modifies what is being communicated by inserting, deleting or changing the data being communicated. If a MITM attack is successful, the attacker can gain access to the data that is being protected and may also have the opportunity to change the meaning of what is communicated xi, 54, 55, 63

NIST The USA National Institute of Standards and Technology 6, 8, 26, 50

nonce In cryptography, a nonce is an arbitrary number that can be used just once in a cryptographic communication. It is often a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks 2, 12

P2P Peer to Peer (P2P) is a network topology that allows arbitrary devices to talk to one another. P2P does not rely on the typical client to server paradigm 64

PFS Perfect Forward Secrecy (PFS) ensures that only future authentication can be affected by long-term secret compromise and that past session keys will not be compromised 54, 55, 57, 63, 66

PII There is no formal agreed-upon definition of Personally Identifiable Information. Informally PII is information that can be used to distinguish or trace an individual's identity 15, 21

Preimage Resistance Preimage Resistance is the property that a hash function computationally infeasible to recover the input (x) given the hash value $h = Hash(x)$. Preimage resistance assumes that the entropy of the x is high enough to avoid a brute force search 67

PRNG Pseudo Random Number Generator, an algorithm that takes a seed or other entropy and stretches it into a longer set of random numbers 5

protocol freshness A guarantee that the protocol is not a replay of either all or part of previous communication 1

RNG Random Number Generator 51, 66

SNI Server Name Indication (SNI) is an extension to the Transport Layer Security (TLS) computer networking protocol by which a client indicates which hostname it is attempting to connect to at the start of the handshaking process. This allows a server to present one of multiple possible certificates on the same IP address and TCP port number and hence allows multiple secure (HTTPS) websites (or any other service over TLS) to be served by the same IP address without requiring all those sites to use the same certificate [39] 11, 22, 35

Strong RSA assumption The SRA is the assumption that the following problem is hard to solve. Given a randomly chosen RSA modulus n and a random $z \in \mathbb{Z}_n^*$ find $r > 1$ and $y \in \mathbb{Z}_n^*$ such that $y^r = z$ Note that this differes from the ordinary RSA assumption (RA), in that for RA, the exponent r is chosen in a way that depends on Z. The SRA is a potentially stronger assumption than the RA, but at the present time, the only known method for breaking ether RA or SRA is to solve the integer factorization problem [73] 46, 48

TLS Transport Layer Security (TLS), The dominant protocol to secure web and other traffic on the Internet 1, 41

Ме́ньше зна́ешь – кре́пче спишь.

Russian Proverb

Chapter 1

Introduction

Most people today cannot imagine a world without eCommerce. Applications from shopping sites to business banking would not be possible without secure Internet communications. The vast majority of this trust has been placed in the Transport Layer Security (TLS) protocol [1]. Is the TLS protocol as secure as it can be?

Breaking into any system requires knowing a few weaknesses; TLS is no exception. These weaknesses, individually, are not usually perceived as a risk, but if the attacker can gather enough weaknesses, the security of a system can be breached. This dissertation focuses on one of those weaknesses to determine if it is real and what can be done about it.

To assure us that TLS is safe, cryptographers have proven that the TLS *protocol* is secure [2]. A protocol is like a recipe, and the proof is a guarantee that the recipe is correct. Cryptographers will say, "Insert random number here," which makes the proofs of protocol freshness simple. Given that TLS *protocol* is proven secure, how can we be sure that the *implementations* are secure?

We know that there are many implementations and versions of implementations of TLS. Prominent vendors like Microsoft and others have their implementations and their armies of developers and testers, but those are not the only implementations out there. There are both proprietary and open-source implementations that are typically used for embedded systems and IoT devices. Are they all correct?

TLS uses random numbers for two purposes, secret key material and public nonces. Secret key material is the only information that keeps your data from attackers. This secret key material is vital to the privacy of the communications. TLS uses nonces as public random numbers to protect the freshness of the conversation. By asking for new random numbers from the participants, the protocol can guarantee the messages are not old messages being replayed. In theory, the key material and nonces are unrelated; they are both simply random numbers. In practice, these two items usually come from the same random number generator. If this random number generator is flawed, both the public random numbers and secret keys are suspect.

The TLS standard specifies that the nonce is 32 bytes, with the top 4 bytes potentially being the time of day while the bottom 28 bytes (224 bits) must be random. One way of thinking about the randomness of the nonce is to imagine a bucket containing 87 six-sided dice poured out and placed in a line. Mathematicians know that if we did this step two billion times, the probability that any two sequences are the same is so close to zero that it may well be impossible. The lack of repetition does not guarantee that a number is random, but if a single duplicate nonce occurs, it is far more likely that the repeating nonce is not a random number.

Nonces weaken TLS because they scream "look here" to values that are not random. Attackers can passively watch the traffic to identify insecure implementations. Once an insecure implementation is found, it can be attacked. In the best case, implementations using bad random numbers have no proof that the implementation is secure. In the worst case, the implementation provides no security.

Intentionally and unintentionally insecure random numbers generators threaten the security of TLS.

Intentionally insecure random number generators have been discovered and publicized. A device produced by Crypto-AG starting in the 1970s used a random number algorithm that was *"created by the NSA, which could therefore decrypt any messages enciphered by the machine"* [3]. More recently, *"NSA's backdooring of Dual EC* [random number generator] *was part of an organized approach to weakening cryptographic standards"* [4]. After much discussion, these systems have been shut down.

Unintentionally insecure random number generators may be worse than intentionally insecure random number generators. Insecure random number generators can make a product perform some sensitive function allowing anyone who knows the flaw to recover the data. One example of unintentionally insecure random number generators is bad random numbers used to create RSA keys. When insecure implementations are found, the vendor may have gone out of business or, worse, are either unwilling or unable to fix their devices. We are concerned that some developers do not care and continue to sell their insecure products to an unsuspecting public. Could the IETF's cryptographic protocols be designed to make it harder to get unintentionally wrong?

Starting with an article in the New York Times Feb of 2012 [6] there has been a constant stream of academic papers documenting that the situation is getting worse, not better.

- 2012: Researchers discovered that *"two out of every one thousand RSA moduli that we collected offer no security,"* which was caused by RSA private keys not being random. A Greatest Common Divisor (GCD) was run against known public keys, which resulted in breaking 26,965 of these keys [7]. The root cause was a *"boot-time entropy hole in the Linux random number generator"* [8].

- 2016: The number of vulnerable machines swelled to *"over 313,000 keys vulnerable to the flaw"* and *"many vendors appear to have never produced a patch"* [9].

- 2019: The number of keys compromised rose to more than 435,000. The majority of new vulnerable keys keys were attributed to Internet of Things (IoT) devices. *"The widespread susceptibility of these IoT devices poses a potential risk to the public due to their presence in sensitive settings."* [10].

Secure random number generators *can* be built. Many papers, standards, and open-source implementations, of secure random number generators, have been published, yet engineers continue to get it wrong. *"In fact it seems that engineers are not able to get it right and it became a serious problem in cryptology"* [11].

When cryptographers hear that their theoretical proof of security gets implemented using insecure random numbers, their comment is typically tantamount to "not my problem," and they often denigrate engineers as sloppy. The use of public random numbers in protocols is commonplace, but their use is not universally considered a good thing. Some cryptographers, this author included, think that exposing raw random numbers is not a good idea. Moti Yung, who has published papers on kleptography [12] stated "*when open random values and crypto random values intermixed (come from the same generator) it is indeed a bad idea!*" [13].

Reliance on public random numbers provides easy proof of security for the mathematician, but engineers have an almost impossible task to test their implementations for correctness. This dissertation aims to determine if the public random numbers used by TLS are a problem, and if so, can TLS be fixed.

Chapter 2 discusses related work on bad random number generators. Chapter 3 describes the methodology that was used to collect actual data regarding TLS's use of random numbers. Chapter 4 discusses the results that were found detailing each cluster of information. Chapter 5 is a discussion of why the bad random number generators happen and what needs to be done to solve this problem. Chapter 6 describes a potential TLS protocol that does not rely on public random numbers, and mitigates the effects of low entropy random number generators. Chapter 8 summarizes our findings. Appendix A describes a random number generator using image sensors to create true random numbers.

三人行，必有我师。

孔子

Chapter 2

Related Work

We cover the history of known RNG failures and current work in the IETF to make TLS less fragile.

2.1 History of intentional and unintentional RNG failures

H-4605

The H-4605 was the last rotor machine produced before being replaced by transistors and, ultimately, computers. The H-4605 was created by Crypto-AG in the 1970s using a Pseudo Random Number Generator (PRNG) created by the NSA so that the NSA could decrypt messages encrypted with the device. NSA's relationship with Crypto-AG lasted until 2018, when the company was liquidated [3].

Netscape browser

In 1998 two Ph.D. students Ian Goldberg and David Wagner, decided to look at the random number generators of this new thing called the Netscape Browser [14]. They were not given the source code but could disassemble the code and recover the algorithm used to create the seed for the PRNG. They discovered that the seed only used four values, the Unix time of day in seconds and microseconds, `pid`, and `ppid`. The time of day in seconds is known because that is when the message arrives. All the other bits are, worst case, a 47-bit search space, best case only 10 bits. Considering that the US version of the browser claimed 128 bits of encryption strength, this was seen as a failure.

The security community has painfully learned that small bugs in a security-critical module of a software system can have serious consequences and that such errors are easy to commit [14].

The authors argue that external peer review is needed for any security software so that consumers can have confidence in a vendor's product. We believe that peer review is not sufficient.

Debian

An automated tool called out two lines of code for using uninitialized memory. One of the lines was deliberately using uninitialized memory as an optional source of entropy. The other line of code moved the actual entropy into the PRNG seed. The maintainer of the routine removed the offending lines, which resulted in the only entropy in the PRNG being the 16-bit `pid`. The result was that all the keys generated on Debian machines between 2006-2008 had only 16 bits of entropy. The traffic from Debian machines during this time was not secure, but the real fallout was that all the public keys and certificates created during this time needed to be revoked [15].

The maintainer discussed his changes publicly on several mailing lists, and the change was placed in open source and stayed there for over two years. This problem and how long it existed before being discovered suggests that peer review is not universally successful.

EC DRBG

NSA was successful at an "exercise in finesse" [16] to get a compromised PRNG as an international standard. National Institute of Standards and Technology (NIST) standardized EC-DRBG over the objections of some famous cryptographers and it remained in force from 2006-2013 [17, 18, 19, 20].

Here the standard was created and criticized by the public and was still published and promoted to an international standard.

NXP Mifare Classic RFID tags

Contrary to the previous examples, the Mifare Classic RFID tag was a hardware-based proprietary algorithm that was not publicly disclosed. The Mifare Classic RFID tags were used for over a decade, with over a billion units deployed, primarily as transit cards. Reverse engineering the hardware was accomplished recovering the proprietary algorithm and two flaws. The first flaw was that the key was only 48 bits leading to a rainbow table attack. The second flaw was that the random number generator was an LSFR with only 16 bits and only depended on message timing which allowed the attackers to get the random numbers to repeat. The researchers showed that the MIFARE Classic was not suitable even for low-value financial transactions [21].

Microsoft Windows 2000/XP

A group of security researchers in Israel determined that if the PRNG state could be discovered, past random values could be recovered with 2^{23} work (seconds) and future values $\mathcal{O}(1)$ to compute. The state of the random number generator was kept in user space and was seldom refreshed.

> *These factors demonstrate the importance of providing a secure pseudo-random generator by the operating system. The designers of the operating system can be expected to be versed with the required knowledge in cryptography, and know how to extract random system data to seed the generator. They can therefore implement an efficient and secure generator* [22].

We generally believe this to be accurate, but large operating systems with well-trained programmers can not always be assumed. The IoT market is the most prominent exception to this rule.

PlayStation 3

In 2010 researchers noticed that Sony was using a stuck-at random number generator to sign content using ECDSA. The DSA and ECDSA standards have always known to be fragile in that if the random number repeats only once, the private key can be recovered. In this case, the key exposed was the signing key that ensured that the content run on the PlayStation 3 was authentic. Although Sony tried to sue the people that recovered the private key into silence, the keys are still out there [23, 24].

DSA was given to NIST at the same time that RSA was being standardized. Given that DSA was far more fragile than RSA, there have always been questions surrounding the decision of the NSA to standardize DSA.

> *The decision to select DSA over RSA was not broadly supported by the cryptographic community. Its main advantage (for the government) seems to have been that – unlike RSA – DSA could not be used for encryption; it is hard to see any advantage in this for the users.*
>
> *The DSA is also a brittle standard: even a small bias in the generation of random numbers for a signature results in leakage of the private signing key. It would have been possible to partially mitigate this problem by computing this random number by hashing the private signing key and the message* [25].

We speculate that the reason DSA was proposed may have been to protect against the GCD vulnerability [7] (discussed next) which would not be public for another 22 years. Trying to dissuade the use of RSA, for this reason, is reasonable, as was keeping the GCD vulnerability to themselves. In private conversations with cryptographers involved with China's encryption standards after the publication of [7], they said unequivocally that the GCD vulnerability was the reason that RSA was never certified as a Chinese national standard.

What we believe was not reasonable was proposing DSA with the flaw where a "*small bias in the generation of random numbers for a signature results in leakage of the private signing key*", all while knowing that a deterministic DSA was possible. Literature since then has started to include deterministic DSA variants [26, 27] and has also begun the process of vetting these algorithms for practical uses by analyzing them for side-channel and other attacks [28].

One could argue that giving the commercial cryptography community an algorithm as fragile as DSA is similar to giving a child a sharp knife. Could this be another example of an "exercise in finesse"? Should not NIST be concerned about fixing this flaw in their standard?

RSA public key factoring

In 2012 two independent teams discovered that there were tens of thousands of RSA public keys that share factors. It was commonly thought that at the time, the probability of two identical 512-bit prime factor being chosen at random would be impossible. A GCD calculation was performed over all the public keys to test this theory, and it was shown that 28 thousand public keys shared factors [7]. One of the teams went further to determine that many of these duplicate keys were caused by a "*a boot-time entropy hole in the Linux random number generator*" [8].

The recommendation we find most relevant to this thesis is:

> *Primitives should fail gracefully under weak entropy. Cryptographic primitives are usually designed to be secure under ideal conditions, but practice will subject them to conditions that are less than ideal* [8].

Java nonce collision

In 2013 several wallets on the Android platform were compromised [29, 30] and the source of the problem (incorrectly) was attributed to a Java Nonce collision. The actual flaw was in the boot time entropy hole in the Android operating system [31, 32]. The fix was to make sure that *Android's OpenSSL PRNG is initialized correctly* [30].

GCM repeated nonces

In 2016 a group of researchers scanned the Internet connecting to sites to analyze the quality of random numbers used as GCM nonces.

> *Despite currently being the most popular TLS cipher, AES-GCM is not well received by the cryptographic community. Niels Ferguson described potential attacks on GCM with short authentication tags* [33]*, Antoine Joux published a critical comment during the standardization process of GCM* [34]*, and several other cryptographers recently described GCM as "fragile"* [35].

The team found "184 HTTPS servers repeating nonces" [35]. It is unclear if this number has gone up or down in the five years since this was published.

2.2 Other efforts to make TLS less fragile

Others consider the fragility of TLS to be an issue. Many have created solutions to address nonce reuse in DSA, nonce collisions in GCM, hiding the server's name being accessed, and mitigating low entropy in TLS clients and servers.

Deterministic DSA

DSA is based on ElGamal, and the vulnerability of nonce reuse was clearly stated in the original paper, "*If any* [nonce] *is used twice in the signing, then the system of equations is uniquely determined and* [the private key] *can be recovered. So for the system to be secure, any value of* [nonce] *should never be used twice*" [36].

This issue has been discussed, and solutions found that generally hash the private key with the message to create a deterministic nonce [37, 26, 27, 25]. Bernstein published a valuable list of papers that have proposed a deterministic nonce [26]. Hashing the ciphertext and the private key means that the only way to get a duplicate nonce is to have the same message or a hash collision. If the message is encrypted, the hash should be of the cipher-text. If the resulting signatures are identical, the message or cipher-text are also identical and the attacker does not gain any information.

Nonce-misuse resistant authenticated-encryption (AES-GCM-SIV)

GCM nonce reuse is as catastrophic as key reuse in a one-time Pad. Arguments highlighting the fragility of GCM have been around since before it was standardized [33]. The proposal for nonce-misuse resistant authenticated encryption (AES-GCM-SIV) states that "*both privacy and integrity are preserved, even if nonces are repeated*" [38].

This mode of operation has the additional benefit: If the nonce is a constant, the only vulnerability is that the attacker can determine if the same message was sent repeatedly. Implementations that have a full entropy nonce allow the security bounds to be higher than the birthday bounds.

The idea of the paper is that the system is still secure even if the nonce repeats. Additionally, if the implementations provides a fully random nonce, the application will enjoy even higher security.

Leaking Destination

Current designs for multi-tenancy for servers and data centers rely on the server name being sent in the clear in a TLS Server Name Indication (SNI) [39] option field. Transmitting the SNI in the clear means that passive Internet monitoring can discover the server name. A document describing Server Name Identification (SNI) Encryption in TLS [40] discusses ways to solve this problem. One method. "HTTP fronting solution" can be used when all of the TLS services are organizationally related.

An example could be Google, where their services are behind HTTP proxy front ends. The initial connection could have an SNI of `google.com`, but inside the protocol, they could provide further differentiation such as wanting to open a connection to `talk.google.com`. Our NAXOS–TLS protocol could be used in this case; see Section 7.

Randomness Improvements for TLS

The Crypto Forum Research Group (CFRG) [41] inside the Internet Research Task Force (IRTF) [42] has published a document that wraps the random number generator to add additional entropy [43]. Their idea is to mix the secret key with the results of the RNG to ensure that there is unguessable entropy in the resulting random numbers. Other than mixing other values in with the local secrets, the TLS protocol is unchanged. We believe they might be misattributing this to the "NAXOS" trick, using the long-term secret key to add entropy was known in 1997 [37] long before the NAXOS paper was published [44].

> "*When we floated these ideas, the TLS 1.3 people said 'we assume good randomness, it's the job of the randomness people' and the randomness people in the CFRG basically said 'your protection mechanisms are out of scope/not needed for our randomness' – we got sent from A to B and back"* [45].

In Chapter 6 we discuss an alternative that replaces the entire TLS Key Agreement Protocol based more completely on the NAXOS protocol.

It is like shining a flashlight in a dark room.

Anonymous

Chapter 3

Methodology

We started this endeavor with one conjecture and a fact. The conjecture was that not all implementations of TLS are correct. Given the checkered history of random numbers and the number of TLS implementations, the conjecture seemed to be reasonable. The fact is that the TLS protocol exposes two public random values (nonces), the Client and Server Hello Random values.

The TLS nonce is present in the client and server "hello" messages. In each of these messages, there is a field called "random". To gain access to these numbers, we need to capture massive amounts of actual TLS Internet traffic [46]. Ideas like setting up a dedicated server were thought of and discarded because bringing traffic to the server is difficult. Scanning the Internet would not work because it only gets to the server-side of the conversation. We finally decided to request a feed of anonymized data from the UCSC IT department. The vice-chancellor for information technology and strategic initiatives at UCSC, Van Williams, was approached and he supported the research as long the privacy of UCSC students, faculty, and employees were respected.

3.1 Gathering the data

UCSC, like many other universities and corporations worldwide, employs a Zeek [47] infrastructure to monitor the UCSC network. Copies of the IP traffic are sent to a cluster of machines that reassemble the individual packets into sessions.

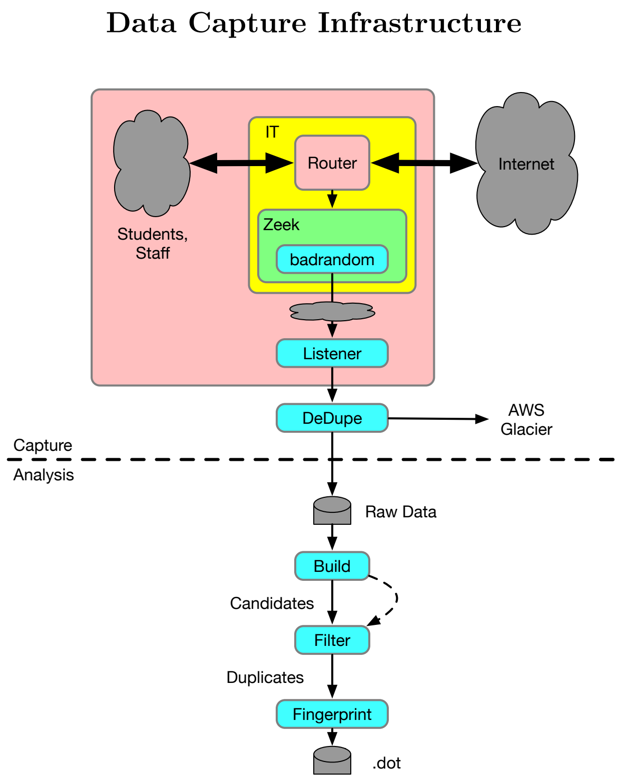

Figure 3.1: UCSC data to and from the Internet is captured by a central router (Router) in the IT department. This traffic is sent to Zeek, which reassembles the streams. A Zeek plugin (badrandom) sends the anonymized information to our server (Listener) for data capture. Duplicates are removed (DeDupe), and the data is fed through a two-step process that creates a Bloom Filter (Build) and potential duplicates and then reruns those potential duplicates (Filter) to ensure we have no false positives. Finally, a graph of duplicates and their implementations (fingerprints) are produced.

The Zeek packages JA3 and JA3S are used to convert the TLS options into a fingerprint by hashing the TLS options in a canonical way [48]. Because there are many TLS options, the probability of two different implementations having the same set of TLS options is negligible.

We created a plugin for Zeek called "badrandom" that is notified whenever a TLS connection is established. This plugin sends the information described in Table 3.1 via UDP to our machine, which checks to make sure that UDP has not duplicated any packets, and stores the information.

Information Captured		
Side	**Name**	**Description**
Client	Fingerprint	Identifying the TLS implementation
	Random	Public random number
Server	Fingerprint	Identifying the TLS implementation
	Random	Public random number
	Server	Certificate desired (SNI)
	IP address	The TCP destination address
–	Time of day	When the connection occurred
	uid	Zeek identifier for this connection

Table 3.1: Fields captured from the UCSC TLS traffic

Once 20GB of data was captured the information was moved to another machine where duplicates were removed. The process looks for duplicate sessions within 1000 records of one another, with all fields the same except for the Zeek identifier. Duplicate entries are caused by problems in the load balancers and other portions of the Zeek cluster and are usually only a few records away. With a span of 1000, no false positives were recorded. The value 1000 was arbiltrarilly chosen based on the number of threads in the Zeek cluster ($\sim$30), the average distance observed was 20, and a hash lookup of size 1000 could be implemented $\mathcal{O}(1)$.

Copies of the raw data were placed in an AWS Glacier storage system to ensure that the results could be recreated in the future.

The random numbers are the focus of the investigation, but the other fields are essential to help us track down the implementation by providing some information about the server that is being accessed. The fields not captured for privacy reasons, such as the source IP address, source MAC address, ethernet port number, and room number, limited our ability to determine the exact device exhibiting problems. Not having this Personally Identifiable Information (PII) did give us flexibility in processing, storing, and sharing. Even with the PII limits, the data we captured allowed us to determine organizations associated with all of the devices that had little or no entropy.

3.2 Analyzing the Data

One cannot determine if an instance of a number is random; random numbers are numbers taken from a random distribution. In mathematics, a group of items that possibly includes duplicates is a bag (also known as a multiset). It is impossible to prove that a bag of numbers came from a random distribution, only that the elements of the bag are indistinguishable from random (which does not mean they are random, but that they look random). The only way to determine if a bag of numbers is not random is to demonstrate that the bag has an implausibly low probability of occurring.

This measure of improbability is related to the entropy in the distribution. Shannon initially thought that entropy should be "uncertainty," which conveys a feeling, but von Neumann ultimately persuaded him to *"call it entropy, for two reasons. In the first place your uncertainty function has been used in statistical mechanics under that name, so it already has a name. In the second place, and more important, no one knows what entropy really is, so in a debate you will always have the advantage"* [49].

Shannon argued that information is the opposite of entropy, but having a bag of numbers indistinguishable from random does not guarantee that they do not contain information. For example, information that is encrypted with an unknown key would be indistinguishable from random unless further action is taken, such as decryption with the proper key.

To determine if a bag of numbers is *not* random requires distinguishers that give evidence of the numbers' non-random nature. The typical tests for random number generators are DieHard [50], and DieHarder [51]. These tests take the input and calculate the probability of various patterns knowing what these patterns should be if the data is random. If a pattern occurs more or less than the expected rate, this is evidence that the numbers are not random. This evidence is said to have distinguished between the actual data and data taken from a random distribution. The more a pattern deviates from the expected the stronger the evidence that the bag does not contain random numbers.

On a bag of data, there is a near-infinite number of patterns that can prove a sequence is not random. Choosing the distinguisher involves trying patterns until one shows a difference. If there are two distinguishers, the distinguisher that shows a lower probability of being random with less data can be considered better.

We noticed early in gathering TLS connection data that there are Client Hello Random values that repeat (the Heninger, Durumeric, et al. team also mentioned seeing repeats [52]). Looking at the goals of the thesis, we decided to focus on these repeats (see Section 3.4) and why they are happening.

The birthday bounds are the expected number of values before at least one of the values has been seen again. The birthday bounds are described as, given a room of 23 random people, more often than not, two will have the same birthday.

In certain bags where the probability a number can repeat is so minuscule, the mere existence of a single duplicate value is an indication that the set has an overwhelming probability that the bag was not drawn from a random distribution. The TLS CLient and Server Hello Random value is 32 bytes in length, but the first four bytes are allowed to be the time of day, which is not random, so we shall concentrate on the remaining 28 bytes. These 28 bytes can contain 256^{28} or 2×10^{67} distinct values. The number of possible values is called population H, and have a similar role to the number of birthdays in a year. We have accumulated approximately one billion (10^9) actual Hello Random values. These actual values are called trials n, and have a similar role to the people in the room. To calculate the probability of one or more collisions, we start with the probability of no collisions. The probability that there is no duplicate random number from a population H after we have seen n values is [53]

$$\begin{aligned} P_0(n; H) &= \frac{H!}{(H-n)! \times H^n} \\ &\approx e^{-n^2/2H} \end{aligned}$$

The probability that there at is least one duplicate random numbers from a population H after we have seen n values is simply the probability that the above did not happen.

$$\begin{aligned} P_{\geq 1}(n; H) &= 1 - P_0(n; H) \\ &\approx 1 - e^{-n^2/2H} \end{aligned}$$

After seeing one billion Hello Random values, the probability of finding a single number that is repeated is 1.85×10^{-50}. Our results found 29,884 distinct Hello Random values, each of which occurred two or more times. Each implementation with a repeated Hello Random value most likely does not take values from a random distribution. The implementations that we have identified are discussed in Chapter 4.

A probability that starts with 50 zeros after the decimal point is a very, very small probability. This is as improbable as someone winning the UK National Lottery jackpot seven times in a row.

As an example, the random value

```
ad100cbcdb10a926acd41f7214d392887472dff54cbd720481b63e15
```

is just as possible as any other value, but repeated even once is evidence that the number was not drawn from a random distribution. In actuality, this value was used as the client Hello Random over 442 times and is the poster child for what could go wrong (Section 4.2.1).

3.3 Locating the duplicates

Post-processing the data was a two-step process. A Bloom filter is used to create a list of duplicates. The duplicates were then used to create the graphical relationship between repeated Hello Random values and the implementations that created them.

A Bloom filter [54] is a space-efficient and high-performance tool to determine if a value is a member of a set. In exchange for the space and performance improvement, the tool has a small number of false positives but no false negatives. The more space that is used, the smaller the false positive rate.

We implemented a three-stage Bloom filter which we populated with the data and output a list of candidate duplicates. Each entry was checked to see if it matched the filter. If the record matched a filter, it was marked as a candidate duplicate. If it did not match, that entry was added to the filter. We eliminated false positives by passing the candidates back through the filter to ensure that every entry matched at least two entries.

Once the confirmed list of duplicates is identified, another pass over the data was necessary to identify the implementations that are associated with these duplicate Hello Random values, and a graph is produced in DOT format [55]. These associations are visualized as nodes for the Client and Server Hello value and the Client and Server implementation with edges that show us the number of times the association occurs.

The entire process is $\mathcal{O}(n)$ runtime and $\mathcal{O}(n)$ space. The original idea was an algorithm with $\mathcal{O}(n)$ runtime and $\mathcal{O}(1)$ space with small false positives but was discarded because of the requirements to keep the data to validate the results. The $\mathcal{O}(1)$ space algorithm may be useful for an informal gathering of duplicates and is discussed in Section 7.

3.4 Hello Random values that did not repeat

This dissertation aims to determine if the Hello Random value used by TLS is a problem, and if so, can TLS be fixed. The repeats show that the Hello Random value is the perfect place to look for egregiously bad random number generators, but what about the "less bad" random number generators that do not repeat?

An example of numbers that do not repeat but are not random could be a counter prefaced by a serial number. Such a number does not have full entropy but also does not repeat. The number of ways people can "do it wrong" leads us to conclude that there are probably random number generators that are not repeating but do not have full entropy. Searching them out may find one or two but will not prove anything about the remainder of the random numbers.

The testing that we have performed is considered “black-box testing”. Black-box testing is where a device is tested without knowledge of how the device under test is constructed. Black-box testing for the random numbers without repeats is possible using tests like Diehard and DieHarder but will require ∼1 billion TLS connections to analyze any one implementation. Getting that much Internet traffic from obscure implementations would be hard.

Analyzing the quality of the random number generators without repeats is best performed as “white-box testing” where the implementation is known. Understanding the implementation would enable both analytical analyses and the generation of the many gigabytes necessary for the Diehard suites.

The Hello Random value used by TLS is a problem for both the egregious bad random numbers that repeat and any bad random number generators that do not repeat because it enables passive monitoring to find vulnerable traffic. To answer the second half, what can be done, we made the conscious decision to focus on the repeats and to determine *why* such an event was happening so we can make a better assessment regarding how to fix the TLS protocol.

Whether or not we find examples of bad Hello Random values that do not repeat is inconsequential since, if we fix the problem for the egregious values that repeat, we will also fix the problem for “less bad” random number generators that do not repeat.

Der Tod eines Menschen: das ist eine Katastrophe.
Hunderttausend Tote: das ist eine Statistik!

Kurt Tucholsky

Chapter 4

Results

We have identified TLS implementations that are not using random numbers generators correctly. These implementations do not meet the assumptions of the proof that TLS is secure.

Correlating the duplicates with the implementations allows us to discover implementations without any entropy (Section 4.2), implementations with low entropy (Section 4.3), and implementations that are intentionally repeating Hello Random value (Section 4.4).

A differentiator that can discover low entropy random numbers that do *not* repeat is possible and discussed in future work (Section 7). We conjecture that low entropy random number generators that do not repeat exist. The existence of these low entropy RNGs does not change our conclusion or recommendations.

Chapter 5 discusses why "naming and shaming" does not fix implementations. The root cause is that the TLS protocol, which in theory is proven secure, in practice is both fragile and has a non-trivial probability of offering no security. A roadmap for fixing TLS is discussed in Chapter 6.

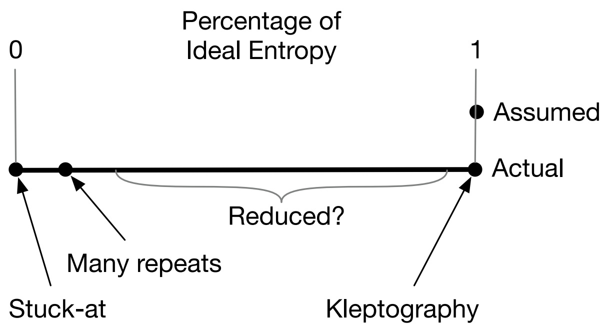

Figure 4.1: The percentage of Ideal Entropy of what was measured is shown relative to what mathematicians assume. Mathematicians assume that there is a CSPRNG which provides ideal random numbers. Actual entropy is a real value and can be from 0 to ideal. Stuck-at implementations have zero entropy. Implementations with many repeats have more entropy. There may be implementations with reduced entropy but do not have repeats (see Section 3.4). The proposed solution (Chapter 6) mitigates the instances where the entropy is less than ideal.

4.1 Visualizing the data

Figures 4.2 and 4.3 graphically show the random numbers that are repeated in our data set. These graphs show 11 separate clusters. Each cluster represents a correlation of implementations with repeated random numbers. Figure 4.2 represents a five-month period September 21, 2020 00:00 GMT to February 2, 2021 04:00 GMT. The Google data (Figure 4.3) was separated because the large number of the collisions required us to limit the period to four hours to make it comprehensible.

Nine of the 11 clusters involve only the Client Hello Random value repeating. One of the 11 clusters repeated both the Client and Server Random value (Section 4.2.4). The one remaining cluster repeated only the server random value and was unable to be verified so that cluster is not included in the analysis.

For the ten clusters that repeated the Client Hello Random values, we tried to discover the client. In a different situation, we would simply ask the user what was the make, model and even the software version of the device that repeated these values, but that was not possible. To protect the PII of the client, the IP address and other identifying information was not provided. We were left to deduce the client device from the information about the server that the device accessed.

The methodology to determine the servers was varied and more difficult because many used Amazon to host their servers. Amazon will not give out the names of their clients but assured us that our Responsible Disclosures notification emails were forwarded to their customers. We have not received any replies from emails that Amazon forwarded.

We know the IP addresses that these clients were accessing. Some TLS connections specified an SNI field in the Client Hello message containing a DNS name we can look up. If the SNI was not present, we tried to obtain the certificate offered by the servers, using OpenSSL to get the "Organization" and "Common Name" from the x509 certificate. If neither of these worked we sent a request with the form:

```
% curl http://<ip-address>:443/
```

expecting a message saying that "The plain HTTP request was sent to HTTPS port". Sometimes the returned message mentions to whom the server belongs. This is not an error, just a way of learning more about who owns a server.

One vendor, Google (Section 4.3.1) knows that repeated Hello Random values are a problem [13] but does not know how to find the device. One might argue that recovering only some of the client implementations is not sufficient. We argue that the results are new and valuable. We identified a previously unknown risk, notified some implementations, and proposed a solution for the clients that we found. We believe that these solutions will also work for those that implementation we did not identify.

The results are divided by implementations:

- Client Random with no entropy.

  Section 4.2.1 CyaSSL, an open source TLS implementation

  Section 4.2.2 An IoT device sending email, Cluster 1

  Section 4.2.3 Superpowered.com

  Section 4.2.4 Vendor [redacted]

  Section 4.2.5 IoT devices accessing Tuya cloud

  Section 4.2.6 Tuya devices padded with variable number of 0s

  Section 4.2.7 TLS scanner

- Client Random with low entropy.

  Section 4.3.1 Google, clusters 0, 9 and 10

  Section 4.3.2 Tuya low entropy

- Implementations intentionally with no entropy.

  Section 4.4.1 Universität Paderborn

Collisions Visualized

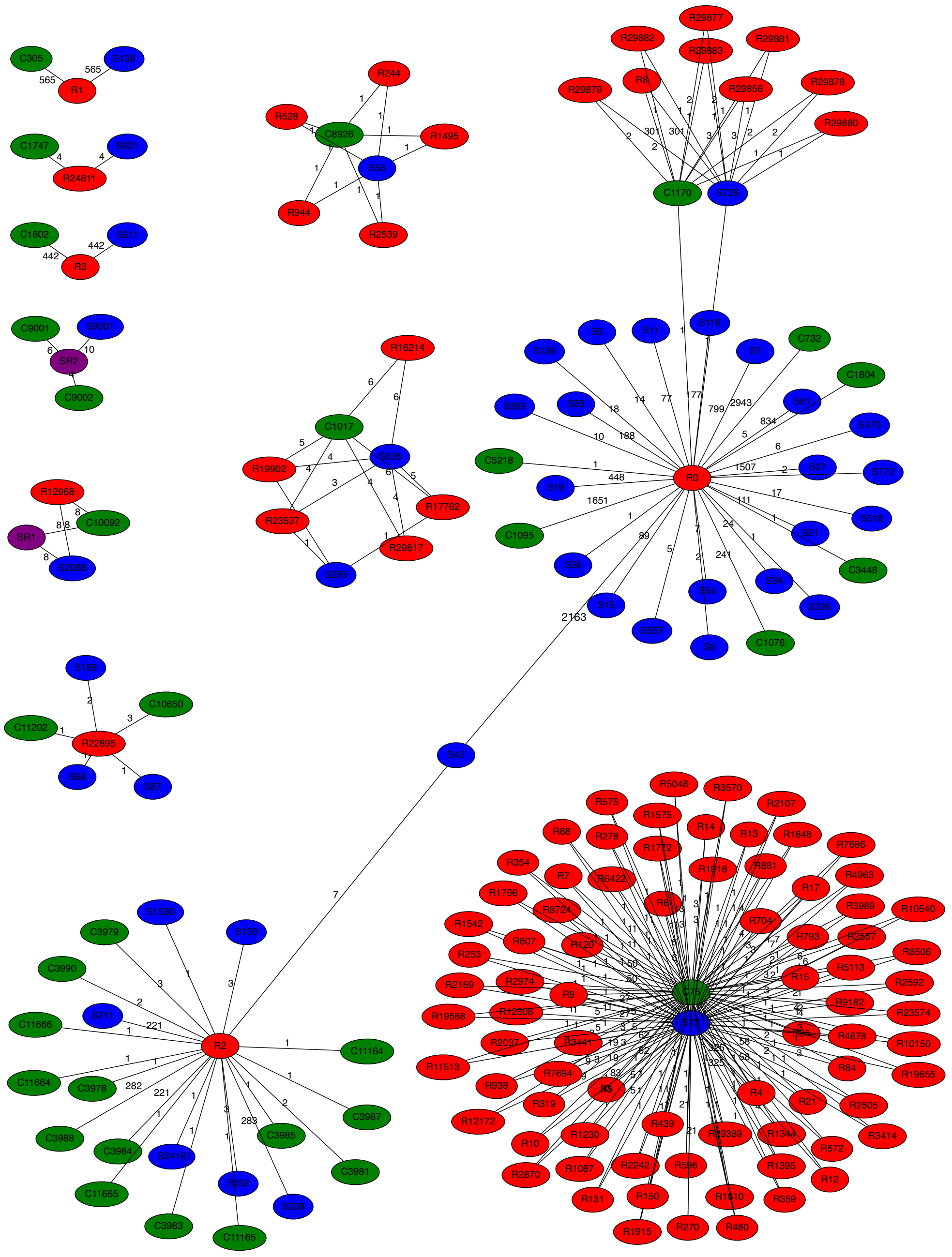

Figure 4.2: Duplicated Client and Server Random numbers (Red and Purple nodes) showing their relationship to the server and client implementations (Green and Blue nodes). The edges show the number of times the relationship occurred. The positions of the clusters and the length of the edges are not significant. Captured from September 21, 2020 12AM GMT to February 2, 2021 4AM GMT.

Google Collisions Visualized

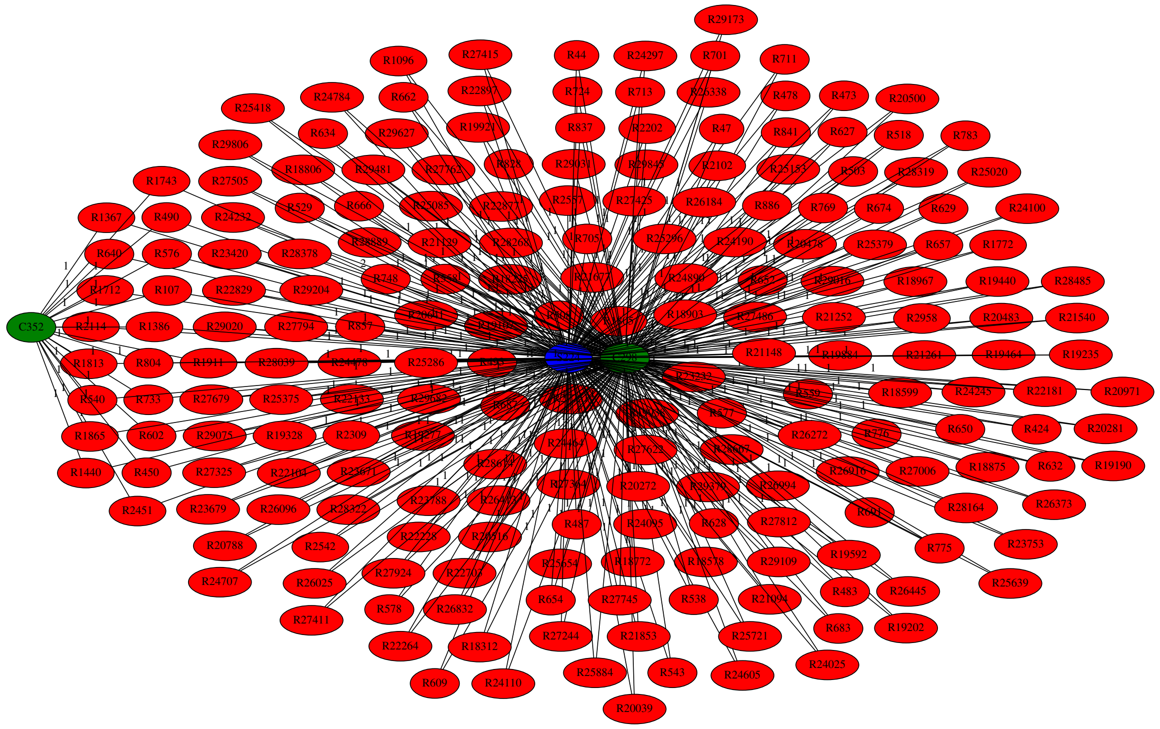

Figure 4.3: The Google collisions from a four-hour subset of the data starting January 15, 2021 6PM GMT. This shows one server (blue oval) and two client implementations (green ovals). These implementations duplicate random numbers (red ovals). This could be one or more devices with the same flaw. See Section 4.3.1.

CyaSSL Cluster

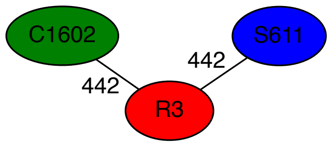

Figure 4.4: CyaSSL Cluster was identified by traffic with a constant random value `ad100cbcdb10a926acd41f7214d392887472dff54cbd720481b63e15`

4.2 Implementations with no entropy

Implementations with no entropy exhibit a single random number for the Client Hello Random value. We refer to these as “stuck-at” random number generators. These system “seed” their random number generator at the beginning of each TLS session, but they do not actually add any entropy. They may have a perfectly good CSPRNG, but without any entropy they just generate the same numbers over and over again.

To our knowledge, all the TLS systems use the same PRNG for both the keys and the Client Hello Random value. This means if the Client Hello Random value repeats, the session keys also repeat. If the implementation is open source (such as CyaSSL below), the code can be instrumented to print the keys whch are the keys that will be used for all past and future TLS connection. Extracting those keys means that the privacy of all past and future communications is lost. It can be said that, in this case, TLS provides no security. Even if the software is not open source, extracting the binary, reverse engineering the code, and recovering the keys is usually possible.

4.2.1 CyaSSL

CyaSSL is an open source TLS Library, so we were able to analyze the source code. The PRNG is based on NIST hash DRBG so it is probably secure if it has been seeded properly. Rather, the failure is that the function to seed the PRNG sets the seed to a constant for certain hardware. The system also reseeds the PRNG at the start of each TLS session, so that the Client Hello Random values and session keys are constant.

GenrateRandom from CyaSSL

```
#elif defined(CYASSL_LPC43xx) || defined(CYASSL_STM32F2xx)

    #warning "write a real random seed!!!!, just for testing now"

    int GenerateSeed(OS_Seed* os, byte* output, word32 sz)
    {
        int i;

        for (i = 0; i < sz; i++ )
            output[i] = i;

        return 0;
    }

```

Figure 4.5: Code copied directly from CyaSSL file `cyassl/ctaocrypt/src/random.c`. This code was repeated 4 times and is used as a place holder for four different sets of hardware, one of which does not output a warning.

In the file `cyassl/ctaocrypt/src/random.c` there are four instances of the code in Figure 4.5. Such a "place holder" or test code accidentally left in is very common in open source projects.

We were able to find three vendors with wildly different products creating exactly the same Client Hello Random value. Table 4.1 lists the vendors we identified as using CyaSSL. In conversation with the owners of CyaSSL, they have now added warnings in the repository that deprecates this implementation [56].

Vendor [redacted]

We were able to get the name of the manufacturer from the certificate on their server, and were also able to confirm the device directly with the UCSC IT department. We have redacted the name because of the nature of the device and because naming the device would add no value to the conclusion. The company acknowledged the problem, shared that they were using CyaSSL [57] and we were able to verify the fix. Their analysis was that *"CyaSSL library function name is GenerateSeed(). This caused* [the] *issue in our case".*

Vendors using CyaSSL with the constant seeds

Vendor	product
[redacted]	[redacted]
Monument	Online Alcohol Treatment
Winix	Air Purifiers

Table 4.1: Vendors using CyaSSL with the constant seeds. Each of these vendors had traffic with exactly the same Client Hello Random value and keys. Traffic with constant Client Hello Random values offers no security.

[redacted] asked if there are tests to cover this kind of failure, and there are none that we know of. Sites that check the clients, such as https://www.howsmyssl.com and https://clienttest.ssllabs.com could be enhanced to test for stuck or repeating Client Hello Random value.

On Oct 19. 2021 [redacted] notified us that the device was fixed and we were able to confirm that the Constant Hello Random value was no longer occurring.

Monument

Some additional traffic with the same Client Hello Random value had offered a certificate that pointed to `*.qa.joinmonument.com`. This site is *"A holistic online alcohol treatment program, tailored to your needs and preferences."* We sent Monument a Responsible Disclosure asking for more information about the implementation. Their response from Kevin, Senior Manager of Care Coordination, Team Monument was:

> *"The issue has not yet been resolved, but it will be something they are working on. As I stated, it is not high priority given that it is a Quality Analysis site that does not have any affect to our member base or our production environment."*

While they did not provide additional information, this traffic did stop being produced.

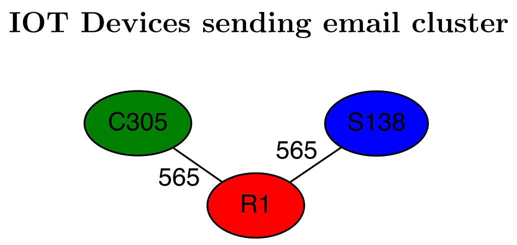

Figure 4.6: IOT Devices sending email cluster with constant random value.

Winix

The third vendor with the same exact Client Hello Random value is Winix. Winix describes itself this way: *"For almost 50 years, Winix has been developing and producing Healthy Home Appliances to bring clean air, clean water and comfort to your indoor environment."* The traffic was identified by the common name in the certificate, `CN=*.api.winix-iot.com`.

There was also traffic to the web site `us.api.winix-iot.com` with Client Hello Random values of

```
0000000000000000000000000000000000000000000000000000000000000000
fbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfbfb
```

A responsible disclosure was sent June 16, 2021 with no response.

4.2.2 Device sending e-mail

In Figure 4.6 the apparently random number

```
a4c5d4d98e7146bc87357bcdceebae08437fc6acd0fe858a3ba64f6b
```

is repeated. This traffic was seen going to both `gmail.com` and `yahoo.com`. Because there are many email clients that can use TLS correctly, this implementation seems like an outlier. We do not know what devices these are and hope that other organizations with more information might be able to identify them.

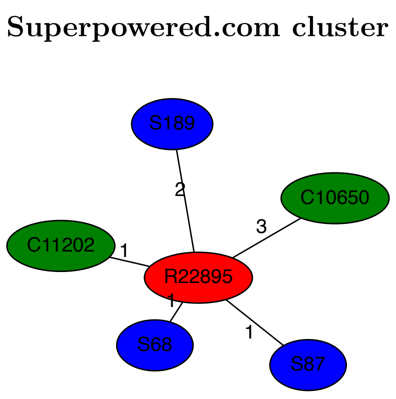

Figure 4.7: Superpowered.com cluster with constant random value.

4.2.3 Superpowered.com Networking Library

This implementation has a constant Client Hello Random value of

`130ff880780bcc3c24b178309d2901f33506ab99a7da23d6b3b0c4dc`

This implementation has a Device fingerprint

`52da4266cab0c4cbbcc34ce82ad336f5`

The Superpowered Networking Library *"Communicate[s] with REST APIs directly via an easy API, bypassing the operating system's HTTP/HTTPS networking stack."* The Responsible Disclosure was sent and Gabor Szanto, Creator of DJ Player, CTO at Superpowered replied that:

> *"yes, our networking library is old and outdated. The aforementioned traffic is a non-critical, optional occasional check on our website and is only https to allow it working from web audio contexts. Before web audio it was pure http."*

Beatport.com

This implementation has a Device fingerprint

`52da4266cab0c4cbbcc34ce82ad336f5`

It is suspected that Beatport.com is also using the same Superpowered.com library. The responsible Disclosure was sent with no reply. Superpowered.com has also been notified that Beatport traffic may be vulnerable, also without reply.

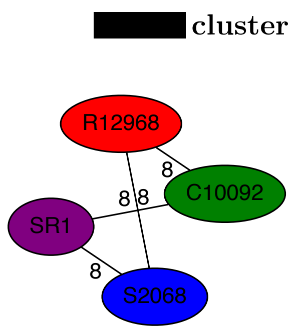

Figure 4.8: [redacted] cluster where both the Client and Server Hello Random values are the same.

4.2.4 Vendor [redacted]

This is unique in that it uses a constant Client and Server Hello Random value. The destination IP address was [redacted] Although we were not able to reverse lookup this IP address, `whois` reported that the address range [redacted] belongs to [redacted]. A responsible disclosure was sent and acknowledged. No results at this time.

Tuya (1 of 3)

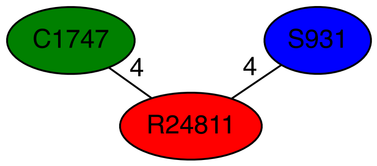

Figure 4.9: Tuya (1 of 3) with constant random value.

4.2.5 Tuya (1 of 3)

A constant client random number

`0870ca0ef1ed3409b47fc675ddd931a46fc0261150b4198f3b9a1bde`

Sending an unencrypted request to the https port on the server identified the machines as `gateway/tuya-psk2`

The response to the Responsible Disclosure by Joy Liu 帝林 was:

> *"After confirmation by the internal technical team, Tuya upgraded the random number generator of the smart terminal in April 2019. The previous implementation method internally confirmed that there is indeed a security risk. In August of this year, Tuya's product baseline and SDK strengthened the function of random number generation. At present, the random numbers of Tuya SDK and modules are gradually replacing the old pseudo-random number generation algorithm with strong pseudo-random number or true random number generation. Since Tuya is a solution provider, whether the specific product is patched or not requires the customer to actively go to Tuya's IoT Platform to update the product's firmware, or authorize Tuya to assist in completing the upgrade."*

4.2.6 Tuya (2 of 3)

This device has a ja3 fingerprint of `319a100e223e7586d58b4379a567cb44` and sent the following Client Hello Random value

```
42416f68626d6436614739314946523165566a614735766247395342
```

hundreds of times. There are also occurrences with trailing 0s, which are itemized with counts in Table 4.2.

All of these random values indicate that there is a problem. The response to the Responsible Disclosure is shown in Section 4.2.5.

Tuya (2 of 3) table of repeated values

count	values
23	00000000000000000000000000000000000000000000000000000000
28	42000000000000000000000000000000000000000000000000000000
37	42410000000000000000000000000000000000000000000000000000
27	42416f00000000000000000000000000000000000000000000000000
29	42416f68000000000000000000000000000000000000000000000000
17	42416f68620000000000000000000000000000000000000000000000
20	42416f68626d00000000000000000000000000000000000000000000
24	42416f68626d64000000000000000000000000000000000000000000
29	42416f68626d64360000000000000000000000000000000000000000
35	42416f68626d64366100000000000000000000000000000000000000
29	42416f68626d64366147000000000000000000000000000000000000
26	42416f68626d64366147390000000000000000000000000000000000
29	42416f68626d64366147393100000000000000000000000000000000
28	42416f68626d64366147393149000000000000000000000000000000
24	42416f68626d64366147393149460000000000000000000000000000
26	42416f68626d64366147393149465200000000000000000000000000
39	42416f68626d64366147393149465231000000000000000000000000
16	42416f68626d64366147393149465231650000000000000000000000
30	42416f68626d64366147393149465231655600000000000000000000
14	42416f68626d6436614739314946523165566a000000000000000000
24	42416f68626d6436614739314946523165566a610000000000000000
38	42416f68626d6436614739314946523165566a614700000000000000
32	42416f68626d6436614739314946523165566a614735000000000000
29	42416f68626d6436614739314946523165566a614735760000000000
27	42416f68626d6436614739314946523165566a614735766200000000
29	42416f68626d6436614739314946523165566a614735766247000000
22	42416f68626d6436614739314946523165566a614735766247390000
30	42416f68626d6436614739314946523165566a614735766247395300
8640	42416f68626d6436614739314946523165566a614735766247395342

Table 4.2: The table of repeated values for Tuya (2 of 3) showing that the most common value repeating does not have trailing 0s, and that every possible number of trailing bytes being 0 has occurred in our data including that all 0s value.

4.2.7 TLS Scanner(s)

This category includes one or more TLS scanners that have a Client Hello Random value of all 0s. We were able to get the IP addresses and were able to determine that they were all located in Sweden, with a network provider of either `GlobalConnect AB` or `GleSYS AB`. Some of these nodes identified in `whois` as `Availo Networks AB`, `Internetbolaget Sweden AB` or `Kortea AB`. This may be one scanner or many. We believe they are TLS scanners because they are walking the public web sites at UCSC and all the traffic is from the Internet.

4.3 Implementations with reduced entropy

We have identified two implementations with reduced entropy. We consider entropy reduced if repeats are happening and it is not "stuck-at" a constant value.

4.3.1 Google

The Figures 4.3, 4.10, and 4.11 all represent one or more implementations that are using Google services. These implementations were identified as Google devices because of the SNI that the traffic requests. These implementations may not have been developed by Google but they are in the Google ecosystem.

```
connectivitycheck.gstatic.com
dl.google.com
embeddedassistant.googleapis.com
fcm.googleapis.com
play.googleapis.com
www.googleapis.com
```

Figure 4.12 shows the distribution and the Table 4.3 describes the most often occurring random numbers.

Google was notified and their response was that *"Matching logs will be difficult, plus not clear we would have data to determine more about the specific client."* , and that we should *"disclose your findings publicly to show that there are devices at risk to better identify the devices"* [58].

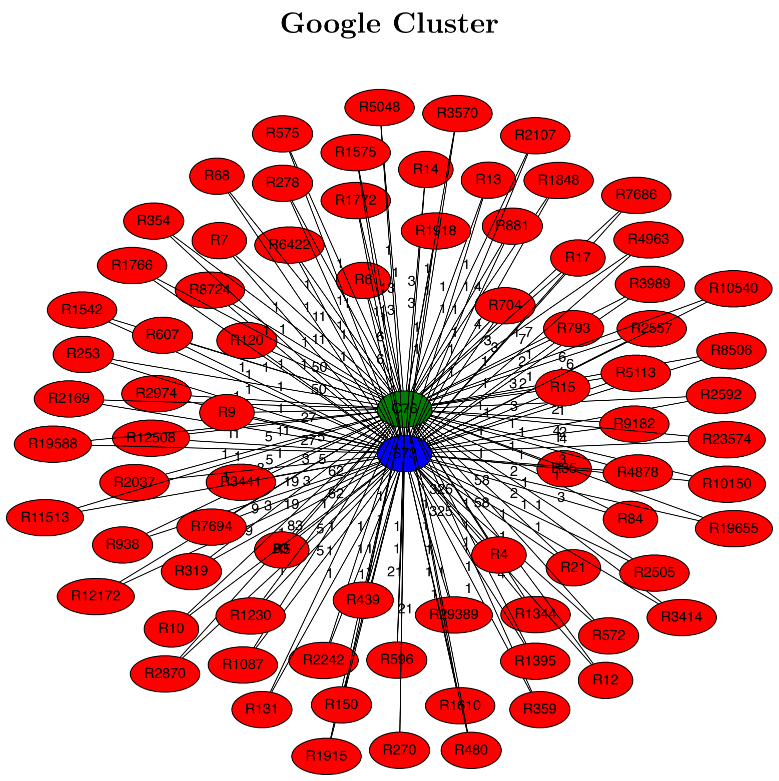

Figure 4.10: Google Cluster around the client and server implementations showing many random repeats.

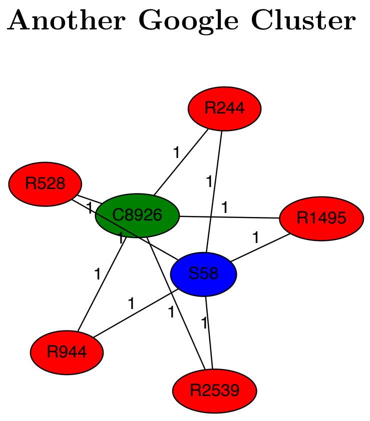

Figure 4.11: Another Google Cluster

Distribution of Google repeats

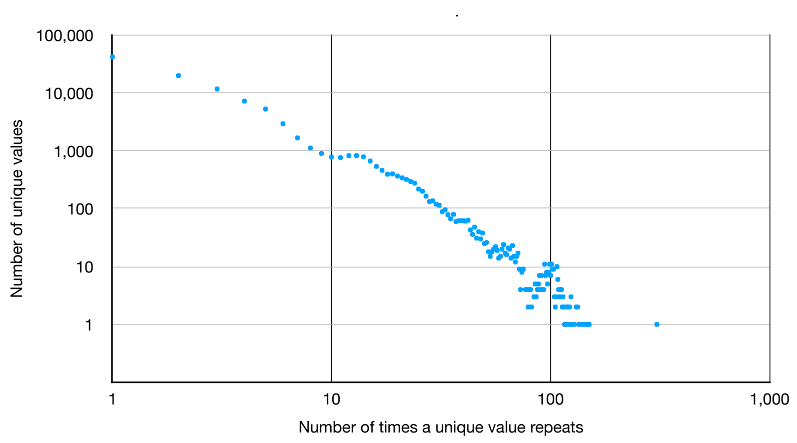

Figure 4.12: For the client implementation at the center of Figure 4.11, there were 462,816 TLS connections. Of these connections, 42,066 had Client Hello Random values that were unique (no duplicate). 1,120 values had 8 repeats. One value repeated 306 times. This result is consistent with a PRNG that starts with a constant seed and diverges over time.

Top 10 Google repeats

count	values
131	cb0a4ad5543414af6272419a1ee3a24e910db36c268e47a18ec1c6e9
131	f93c341695ec701f11e4385b9082deab85f06628a8e9e6c5a7c5b5fa
133	113d24db7ea669712ff88f0697f1a052fd9a81c0aa70b240781a57dd
133	991ca417b1b117742e8a5b1185e1ec66c41d08031228f37d756c4159
134	86a5c9920a0eea8282072b0f004f2d47b8d73af9004db0f846be836b
138	4bd0080ecfd149490b100814e22d199c156f2648fba796a464b6cd79
143	a7470aa4a5158651a0e99bb325522247c28ade2ef8e417ad99806f8c
148	45d3b4a6d0a1ac925fbb6ab38283c5601ecb720f3339a7f06a039c95
150	83dd46b41577fdb6cd730b2d678d2741f4823911c99284d6aeda5112
306	1334d6c52d6d419ecba0869b4b30ab3a0cf83255d5c07858bc98ab5b

Table 4.3: Top 10 (by count) repeated Client Hello Random values for Google

Tuya (3 of 3) cluster

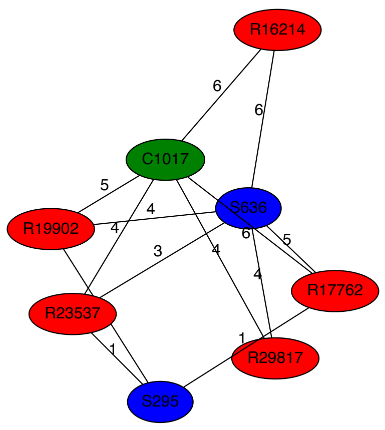

Figure 4.13: Tuya (3 of 3) cluster

4.3.2 Tuya (3 of 3)

This device has a ja3 fingerprint of: `5a998b06b4ed97beb4a9fc62444db543`

This device repeats various ClientHello Random values. The Server Name Indication is to your host: `a2.tuyaus.com`

The top 10 repeated values are shown in Table 4.4 The response to the Responsible Disclosure is shown in Section 4.2.5.

Top 10 repeats Tuya (3 of 3)

count	values
8	2e4daaa10339db09bbd195e1816ae353b38823749a78b35e41630638
9	da22b2b95e1a16756bcfef1fea7def664efce2fd50441a8d1d50e30a
10	f9796b521413e9e22d518e1f56085727a705d4d0528277751b994aed
11	7b26d0c5fdbea337ad066d30e916408f4860e4db1a220c2807400a6b
33	cf445bd48602854262e36161c19807c3daed605df6da83a4697b82c6
36	8c0a84e96c11f355842a84f6686a6fccb9c69f9ef34644fcc3bae58e
42	034ba98a81f18ccd039ef0a18795310525ddc12a9c7eaf1e7450967b
42	69f8a70164ab15095b6b7b8b9cfa117324f00b67acca204c3664a732
49	eca03c6ae2f0c1f578cea9c96641086a0c352cb12ad13cabc5ef5455
283	53bef1fcf9796b521413e9e22d518e1f56085727a705d4d052827775

Table 4.4: Top 10 (by count) repeated Client Hello Random values for Tuya (3 of 3)

Distribution of repeated values, Tuya 3 of 3

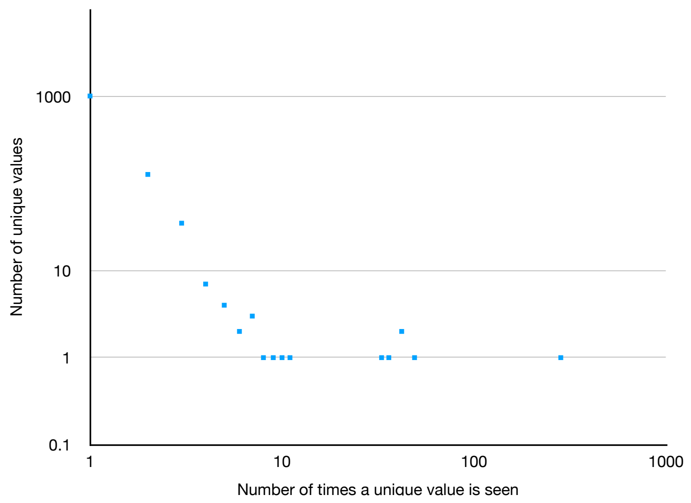

Figure 4.14: Tuya (3 of 3) cluster

4.4 Intentionally constant Client Hello Random values

We were able to discover one implementation that has intentionally used a constant as their TLS Client Hello Random value.

4.4.1 Universität Paderborn TLS Scanner

This implementation has a constant Client Hello Random value.

```
60b420bb3851d9d47acb933dbe70399bf6c92da33af01d4fb770e98c
```

This is the University of Paderborn TLS crawler [59] and intentionally uses constant Client Hello Random value as well as constant secret keys.

> *We do various internet wide scans, sometimes with zmap, sometimes with zgrab and more often than not with our own TLS-Attacker based scanner (the large scale scanner is not public accessible at the moment). For the most part, our scanners use static randoms. Our scanner goes even further and uses static keys for everything. We sometimes need to change it up, but for the most part we keep it static. Since our scanners do not transmit any sensitive data security is not a concern for us. Our scanner is actually fingerprinted by some snort rules (https://www.snort.org/rule_docs/1-45831), therefore changing it would bypass the IDS of some companies. We respect their wishes to be "not scanned" and therefore keep it static for the most part such that they can fingerprint and block us (if they wish so)* [60].

Fragility, like the complete loss of authentication security, is not a desirable property.

Niels Ferguson

Chapter 5

Discussion

TLS is a black box cryptosystem because only the developer knows what was compiled. Some believe that open source software solves this problem because the code can be examined and that *"constant peer review creates more secure applications"* [61], but this is not the case. 84% of the open-source codebases analyzed had at least one vulnerability, with an average of 158 vulnerabilities per codebase [62]. There have been cases of individuals putting intentional backdoor [63]s or bugdoor [63]s into open-source software. Even if the codebase is secure, the user does not know if the software was changed from the public version.

Over the years, the question of trusting black box cryptographic systems has been discussed, but usually in the sense of trusting a government-supplied black box cryptographic system. Moti Yung, who has published books on malicious Cryptography [64] states:

> *Some of the most important questions that arise with respect to black-box cryptography are the following: Is the algorithm contained within the black-box secure? Does it leak secret key information? If someone ever successfully reverse-engineers the black-box am I at risk* [65]?

But what about TLS? The TLS protocol did not come from the government, so *can* it be trusted? We have discovered that the TLS standard makes identifying implementations easy and finding bad random numbers trivial. We have shown that the TLS standard is complicated to "get right" and can result in implementations that offer no security. We argue that the TLS standard is fragile and leave the question of why to future work (Section 7). The remainder of this document will concentrate on what can be done to make TLS less Fragile.

We discuss why notifying vendors is not enough, why TLS should not make implementation details accessible, and why the protocol should not provide access to raw random numbers. We then recommend "fixing" TLS by proposing a new version that mitigates the amount of information that TLS passively leaks while still allowing clients and servers to be actively tested (Chapter 6).

5.1 IoT devices

A good definition of IoT:

> *Internet of Things (IoT) is a ubiquitous network of uniquely identifiable things, real or virtual, that communicates a massive amount of data to be used for intelligent decision making.* [...] *IoT systems provide a large spectrum of services such as Intelligent Transportation Systems (ITS), smart grids, smart buildings, smart cities, e-health, intelligent drug delivery system etc. Even the Cyber-Physical Systems (CPS) such as Nuclear Power Plant (NPP) comes under the IoT umbrella. Most of these services are critical in nature* [66].

Low-cost processors and communications enable IoT to be able to gather and process information as well as autonomously control complicated systems. To enable the Internet of Things ecosystems, many have been working on Lightweight Cryptography, including the NSA creating the algorithms Simon and Speck.

> *In recent years, computational tasks have progressively been pushed down to smaller and smaller devices. It has been recognized that traditional cryptography is not always particularly well-suited to the needs of this emerging reality.*
>
> *Lightweight cryptography seeks to address this by proposing algorithms and protocols explicitly designed to perform well on these constrained platforms.* [67].

We believe that Simon is a valuable teaching aid for the teaching of block ciphers and, when implemented at full strength, is as secure as AES. That said, any symmetric algorithm with 32-bit block size and a 64-bit key is not secure. Offering these algorithms up as secure for IoT is disingenuous and only provides "security theater". Giving IoT engineers inadequate tools in the name of limited processing power can allow engineers to create devices that lack adequate security. The belief that engineers can use these lightweight cryptographic algorithms because they choose a low-power processor is a non-sequitur and could be seen as intentionally muddying the waters on what it means to be secure. The manufacturers' choice of processor is not an excuse to provide inadequate security.

History has shown that IoT is implemented not by large well funded, well-trained, and security-conscious organizations but is built on open-source software and minimal engineering, often resulting in significant security failures.

> *IoT devices are riddled with vulnerabilities and design flaws. In consequence, we have witnessed the rise of IoT specific malware and botnets with devastating consequences on the security and privacy of consumers using those devices. Despite the growing attacks targeting these vulnerable IoT devices, manufacturers are yet to strengthen the security posture of their devices and adopt best-practices and a security by design approach* [68].

What is related to risk is that some smaller processors lack hardware random numbers generators and may have trouble getting full entropy random numbers. Section 6 discusses how we can help clients that do not have access to hardware random number generators.

5.2 Vendor Notifications

Can we be vigilant and notify the bad implementations we find, or do we have to change the protocol? We have notified the insecure TLS implementation vendors, but is that enough?

The engineers that create these cryptographic implementations are human. There will continue to be bugs until we can get provably correct code for the protocol, applications, and OS. After one high visibility disclosure where 20,000 RSA keys were discovered to offer no security, and the vendors disclosed [8], after four years:

> *We find that many vendors appear to have never produced a patch, and observed little to no patching behavior by end-users of affected devices. The number of vulnerable hosts increased in the years after notification and public disclosure, and several newly vulnerable implementations have appeared since 2012. Vendor notification, positive vendor responses, and even vendor-produced public security advisories appear to have little correlation with end-user security* [9].

In a perfect world, the fix to the vulnerability would be shared with the person reporting the problem to verify the fix, but this seldom happens. Many vendors do not acknowledge the problem; some do not have a way of fixing the problem. Even when the vulnerabilities are sent to the vendors, years later, the number of vulnerable devices is still increasing [10]. In this survey, we were able to verify that one vendor fixed their product. A better solution is to create a cryptographic algorithm that is less fragile (Chapter 6).

Three facts became evident from this survey. First, IoT devices dominate the list of vulnerable implementations. Second, the large number of options defined by the TLS standard and passed in the clear makes it possible to fingerprint specific implementations of TLS with high reliability. Third, the TLS standard makes it easy to identify bad random number generators by analyzing the Hello Random values. The fingerprint and Hello Random value are needed to find low entropy implementations.

5.3 What does it mean for a protocol to be "fragile"?

In mathematical terms, the proof is proof. The result is binary, proven or not. The concepts of better or worse are, in some senses, meaningless. A *better* proof may be more subjectively elegant but does not change the fact that there is or is not proof. What does it mean for proof to be fragile?

The term fragile and fragility have been used in cryptography papers over the years without definition [33, 69, 70]. We will define the term and discuss how this term can be used to compare two proven secure protocols. In general, we agree with the following even though the term *fragility* is not defined.

Crypto is by far the most extensive and repeatedly surprising study of remarkable consequences of intractability assumptions. Furthermore, these consequences (and thus, the assumptions needed for them) form the current foundations of computer privacy and security and electronic commerce, which are used by individuals, companies, armies and governments. One would imagine that society would test these foundations and prepare for potential ***fragility*** *at least as well as it does for, say, nuclear reactors, earthquakes and other potential sources of colossal disasters* [70].

We will start with the definitions of fragile, a related word tenuous and a near antonym robust [71]:

fragile adjective
frag·ile | \ˈfra-jel, -jī(ə)l \

Definition of *fragile*

1 a : easily broken or destroyed
// a *fragile* vase
// *fragile* bones
b : constitutionally delicate : lacking in vigor
// a *fragile* child

2 : TENUOUS, SLIGHT
// *fragile* hope
// *fragile* coalition

When describing a fragile proof that a protocol is secure, sense 2 "tenuous" seems the closest. A proof is a statement that something is true; an assumption is a statement that believed to be true. The proof of a secure cryptographic protocol contains assumptions. For instance, it is common for people to quote the Decisional Diffie Hellman assumption (DDH) [72], or the Strong RSA assumption [73]. These assumptions used to be considered strong, and now in the face of Quantum computers, are considered less so. We choose tenuous because it leverages "strength".

tenuous adjective
ten·u·ous | \ˈten-yə-wəs, -yüəs \

Definition of *tenuous*

1 a : having little substance or strength : FLIMSY, WEAK
// *tenuous* influences
b : SHAKY sense 2a
// *tenuous* reasons

2 : not thick : SLENDER
// a *tenuous* rope

3 : not dense : RARE
// a *tenuous* fluid

Tenuous assumptions are assumptions with little or no basis to believe true but can not be proven false; these assumptions are considered weak. Strong assumptions are not new, but we extend the concept of strong to protocols.

robust adjective
ro·bust | \ˌrō-ˈbəst, ˈrō-(ˌ)bəst \

Definition of *robust*

1 a : having or exhibiting strength or vigorous health
b : having or showing vigor, strength, or firmness
// a *robust* debate
// a *robust* faith
c : strongly formed or constructed : STURDY
// a *robust* plastic
d : capable of performing without failure under a wide range of conditions
// *robust* software

2 : ROUGH, RUDE
// stories ... laden with *robust*, down-home imagery
– *Playboy*

3 : requiring strength or vigor
// *robust* work

4 : FULL-BODIED
// *robust* coffee
also : HEARTY
// *robust* dinner

5 : relating to, resembling, or being a specialized group of australopithecines characterized especially by heavy molars and small incisors adapted to a vegetarian diet

Our definition of **fragile** and **robust** is: Given two proven correct protocols P_1 and P_2 whose assumptions are $a(P_1)$ and $a(P_2)$ respectively, $a(P_2) \subset a(P_1)$ where $a(P)$ is the set of assumptions of the protocol. P_1 is said to be objectively more **fragile** that P_2 and P_2 is said to be objectively more **robust** than P_1.

In 2008, two vendors were creating encrypted tape devices. Both were using AES for bulk encryption. Each chooses a different method to save the key on the tape. One chose an AES key wrap, and the other chose an RSA-based Key Encapsulation Mechanism (KEM) [74]. At the time, both were considered secure. One vendor only assumed the security of AES, and the other assumed the security of both AES and RSA. Objectively, the protocol that only used AES was more robust than the AES and RSA protocol.

Many times protocol proofs rely on stated assumptions like DDH or Strong RSA assumption, but many times they make assumptions that are never stated like $\{0,1\}^\lambda$ is a Cryptographically Secure Pseudo Random Number Generator (CSPRNG) that has been reliably seeded with more entropy than λ. Mathematicians will say, "of course," but the outcome can still be that a TLS implementation of a proven secure protocol offered no security. The fact is that the assumption did exist even if it was not stated.

If two protocols are not strict subsets, the different assumptions need to be evaluated, and one protocol can be considered subjectively more fragile or subjectively more robust. As an example, an obvious but never-stated assumption is that the protocol can be faithfully implemented. Evaluating this assumption between two protocols would mean that the simpler protocol is more robust.

Sometimes issues become apparent after the protocol is proven secure. For instance, a key agreement protocol that leaks the implementation is an unstated assumption that leaking that information is not a problem. Mathematicians can claim that such leakage is "beyond the scope" of the proof.

5.4 Identifying implementation is dangerous and unnecessary

In military jargon, the term "kill chain" is related to the structure of an attack. Cyber attacks are the same. "*A kill chain is a systematic process to target and engage an adversary to create desired effects.*" [75] and the first step is:

> *"Reconnaissance – Research, identification and selection of targets, often represented as crawling Internet websites such as conference proceedings and mailing lists for email addresses, social relationships, or information on specific technologies"* [75]

The implementation identification can be traced to the fact that TLS has so many options that the probability of any two implementations having the same set of options is negligible. The Zeek package JA3 [48] takes advantage of this fact by hashing up a canonical representation of all the options to create a shorthand that identifies both client and server implementations.

The ability to differentiate between implementations is a capability that cuts both ways. The JA3 fingerprint can identify the Trickbot and Emotet malware, which can easily be seen as "good", but JA3 also identifies ToR clients, which could cost human rights workers their lives and be seen as "evil".

As a result of this thesis, we have discovered that an implementation with the JA3 fingerprint of:

```
074977ad77b1d230841d9211d2927611
```

is a Google compatible device with a poor random number generator.

We have shown that the JA3 fingerprints of TLS are a valuable tool to find insecure implementations. Being able to find insecure implementations is not a goal of TLS. We argue that TLS should not enable passive monitoring to identify a particular implementation.

5.5 Hello Random values are dangerous and unnecessary

We believe the Client and Server Hello Random values are dangerous for two reasons: first, our research has shown that inept implementations are trivial to find and, second, these fields make intentionally leaking key materials trivial.

5.5.1 Hello Random makes bad RNG easy to find

The IETF warns the implementor that if a CSPRNG is broken, "*it may be possible for an attacker to use the public output to determine the CSPRNG internal state and thereby predict the keying material*" and that "*Implementations can provide extra security against this form of attack by using separate CSPRNGs to generate public and private values.*" [76]. This statement is an interesting warning, but the standard relegates this paragraph to an appendix and leaves the engineer to determine if separate random number generators are needed. Maybe this paragraph is intended to assuage the fears of cryptographers that have looked at this protocol. We know of no implementations that have gone to the effort to have separate random number generators, so the value of this paragraph seems to be cosmetic.

The IETF runs on a “Rough consensus and running code” model that values interoperability. The IETF cares about security but cannot enforce security. We believe that the IETF could work with the cryptographers to engineer a less fragile protocol, leak less, and either work securely or not at all.

The companies that create TLS implementations employ engineers to implement their devices securely. Engineers are human, and invariably they make mistakes. History has shown that random numbers are difficult to get right. It is not only the algorithm that generates the random numbers, but the seeds for the PRNG, and even simple programming mistakes copying the random numbers into the protocol.

There is no global governmental organization controlling the Internet. In the USA, there is NIST, but NIST does not standardize the Internet. NIST has published documents on random number generators, but these are just recommendations. National standards like Common Criteria help, but it is not mandated and does not guarantee correctness. Governmental Cryptographic organizations like the NSA are responsible for protecting national infrastructure like the government, banks, electrical grid but do not protect consumers.

Cryptographers use random numbers because it is easy. It is trivial to prove freshness by simply saying, “insert random numbers here.” Engineers find it hard to implement random number generators and have shown that simple mistakes can eliminate the security that TLS provides. If the implementation offers no security, a proven secure algorithm is of little use.

Passively monitoring the Internet traffic that has the Client Hello Random value of

```
ad100cbcdb10a926acd41f7214d392887472dff54cbd720481b63e15
```

means that there is an overwhelming probability that the person monitoring the traffic can decrypt the traffic.

5.5.2 Kleptography

The word kleptography was coined to represent implementations that intentionally leak key materials and other secrets through a covert channel [77]. The Hello Random value is neither designed nor intended to transfer information and has no limits to the values it can contain, and as such, it is a covert channel. *"Two of the most important factors affecting the bandwidth of a covert channel in any operating system or hardware platform are the presence of noise and the delays experienced by senders and receivers using the channel"* [78]. However, the Hello Random value has neither noise nor delay.

If Alice wanted to intentionally leak a 256-bit AES key K to Eve, Alice could give a key K_{eve} to Eve and then use

$$\texttt{random} = e_{K_{eve}}(K)$$

as the Server Hello Random value. If K is indistinguishable from random, the value `random` will also be indistinguishable from random. Alice, Bob, and Eve now have all the information necessary to decrypt the traffic.

The Hello Random field is dangerous because it leaks information about a bad Random Number Generator (RNG) and can be intentionally misused. Creating a replacement for TLS that eliminates the Hello Random field is needed.

Too many of our cryptographic primitives come without blade guards. They're powerful to use but apt to cut someone's hand off, too.

Jon Callas

Chapter 6

Deterministic TLS

TLS is not new; The protocol design started in 1985 [79]. Since then, the protocol has been poked, prodded, and changed from SSL1.0, SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3, each change motivated by a specific threat against the protocol.

We have uncovered two threats to TLS, one which must be fixed and one that should be fixed. We must remove the Client Hello Random value from the protocol. This value is *not* needed and gives attackers raw access to the random number generator. Access to the raw random numbers allows passive monitoring of the Internet traffic to find implementations with reduced entropy even if that random number does not repeat. Proof writers say "fix the RNG", but this is a simplistic solution that has not worked for past issues and will not work for future issues. Giving attackers raw access to the random number generator is a flaw because proven secure Authenticated Key Agreement Protocols (AKAPs) that do not expose raw random numbers have existed years before the first deployed version of SSL.

We argue that allowing fingerprinting client implementations is a flaw since TLS leaks that a user uses specific privacy-enhancing technology and enables passive monitoring of implementations. Corporate security organizations have been using JA3 since 2019 to provide *"higher fidelity identification of the encrypted communication between a specific client and its server"* [48]. This higher fidelity may help define threats to corporations, but we argue that it is a threat to users because attacking implementations also requires identifying implementations with high fidelity.

These threats are real, and fixing them does not reduce the functionality of TLS. TLS allowing passive Internet monitoring to unmask implementations like ToR and TLS enabling finding weak random number generators are not goals of TLS. These flaws threaten the security and privacy of TLS's users and can be fixed to create a more robust TLS Authenticated Key Agreement Protocol (AKAP).

Recently others have noted that *"future protocols should rely on algorithms and constructions that reduce the risk of implementation errors as much as possible"* and to also ensure that *"erroneous implementations are non-interoperable"* [35]. The NAXOS–TLS protocol (Section 6.4) is intended to reduce risks of implementation errors. This protocol is "deterministic" in that random numbers are only used as key material. Additionally, passive monitoring will not determine the implementation or the clients with low entropy. These changes will not stop the ability to test the quality of the random numbers in deployed servers or clients. The protocol is designed to be all-or-nothing because the protocol has been implemented correctly or an error has occurred. We choose the symbol $\perp$ to represent the binary statement that an error has occurred.

Deterministic AKAPs that do not expose raw random values have been known for over 30 years. An early one was the STS protocol [80] published in 1992. A more recent AKAP, NAXOS [44], has a stronger security guarantee than STS and other Authenticated Key Establishment Protocols. We will discuss STS and NAXOS with why they are inadequate to help with low entropy client random numbers endemic in TLS. We will introduce NAXOS++ with protections for low entropy clients and ensure that the algorithms succeed or $\perp$. We will then extend NAXOS++ as NAXOS–TLS that adds features for certificate handling and TLS option passing in a way that does not leak either the identity of the client or the implementation. These methods are proven secure using the assumption that random numbers are random. We will close with residual vulnerabilities of the NAXOS–TLS protocol if the random numbers are not perfectly random.

Basic STS AKAP Protocol

$$\begin{array}{ccc} \mathcal{A} & & \mathcal{B} \\ x \overset{\$}{\leftarrow} \{0,1\}^\lambda & & \\ & \xrightarrow{X = g^x} & \\ & & y \overset{\$}{\leftarrow} \{0,1\}^\lambda \\ & \xleftarrow{Y = g^y} & \\ & \xleftarrow{C_\mathcal{B} = e_K(s_\mathcal{B}(Y,X))} & \\ & \xrightarrow{C_\mathcal{A} = e_K(s_\mathcal{A}(X,Y))} & \end{array}$$

Figure 6.1: The Basic STS protocol extends the basic D-H protocol [81] to authenticate the endpoints and prevent MITM attacks. Setup (not in figure) system parameters the generator g and modulus are communicated as well as both Alice's $\mathcal{A}$ and Bob's $\mathcal{B}$ public signing keys $s_\mathcal{A}$ and $s_\mathcal{B}$. Alice starts by creating an ephemeral private key x and calculates ephemeral public key $X = g^x$. Alice sends the ephemeral public key X to Bob. Bob then does the same creating y. Bob now has the information necessary to calculate the shared secret K. Bob signs the value (Y, X) and then encrypts it with the shared secret K. Bob then sends Y and $C_\mathcal{B}$. Alice now has what she needs to calculate K and decrypt $C_\mathcal{B}$ and verify Bob's signature. Alice now knows that she is talking to Bob. Alice now can calculate $C_\mathcal{A}$ and send it to Bob. Finally, Bob verifies that he is talking to Alice by using K to decrypt $C_\mathcal{A}$ and check Alice's signature and the authenticated key exchange is complete.

6.1 Basic STS AKAP Protocol

The basic STS protocol is a typical Diffie-Hellman Key Agreement Protocol (KAP) [82] with additional messages to ensure that Alice is talking to Bob. The established key is

$$K = g^{x \cdot y}$$

The STS protocol uses four keys, the ephemeral keys x and y, and the long term private keys used for creating signatures and requires three exponentiations on each side. As is typical, the protocol assumes that the keys have perfect entropy. If the implementation of the STS protocol has perfect entropy, the STS protocol also provides Perfect Forward Secrecy (PFS).

In the STS protocol, privacy is guaranteed unless one of the ephemeral keys is compromised. The signature proves authentication by demonstrating that each side knows their respective private keys. Repeating back X and Y proves freshness and defeats MITM.

Authentication is lost if either long-term key is compromised. MITM defense is lost if both long-term keys are compromised.

Low entropy ephemeral keys that repeat will be visible to passive monitoring by repeated X or Y values. Low entropy ephemeral keys that do not repeat are not visible because $X = g^x$ and relationships between subsequent x values are obfuscated by the Decisional Diffie Hellman assumption (DDH) [72].

6.2 NAXOS AKE Protocol

The NAXOS AKE protocol [44], Alice $\mathcal{A}$ has ephemeral key $ek_{\mathcal{A}}$ and long-term secret key $sk_{\mathcal{A}}$ and Bob $\mathcal{B}$ has ephemeral key $ek_{\mathcal{B}}$ and long-term secret key $sk_{\mathcal{B}}$. The author's proof of security allows "an adversary to reveal any subset of the four which does not contain both the long-term and secrets of one of the parties" [44]. If the attacker has both ephemeral keys but not the long-term key of either party, the protocol is still secure. NAXOS provides Perfect Forward Secrecy (PFS); when an attacker gets both long-term keys, but not the ephemeral keys, the past traffic is still private. If the attacker has a long-term key, that person is vulnerable to impersonation. If the attacker has both long-term keys, future conversations between these two parties are vulnerable to MITM attacks.

The algorithm set-up requires Alice and Bob to create long term secret keys $sk_{\mathcal{A}}$ and $sk_{\mathcal{B}}$, and then calculating their long term public keys $pk_{\mathcal{A}} = g^{sk_{\mathcal{A}}}$ and $pk_{\mathcal{B}} = g^{sk_{\mathcal{B}}}$. The identities of Alice and Bob are binary strings $\mathcal{A}$ and $\mathcal{B}$. The public keys and identities are distributed.

NAXOS KAP Protocol

$$\begin{array}{lcl}
\mathcal{A} & & \mathcal{B} \\
ek_{\mathcal{A}} \leftarrow \{0,1\}^{\lambda} & & \\
x \overset{\$}{\leftarrow} H_1(sk_{\mathcal{A}}, ek_{\mathcal{A}}) & & \\
& \xrightarrow{X = g^x} & \\
& & ek_{\mathcal{B}} \leftarrow \{0,1\}^{\lambda} \\
& & y \overset{\$}{\leftarrow} H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}}) \\
& \xleftarrow{Y = g^y} &
\end{array}$$

$$\mathcal{A} : K_{\mathcal{A}} \leftarrow H_2(Y^x, Y^{sk_{\mathcal{A}}}, pk_{\mathcal{B}}^x, \mathcal{A}, \mathcal{B})$$

$$\mathcal{B} : K_{\mathcal{B}} \leftarrow H_2(X^y, pk_{\mathcal{A}}^y, X^{sk_{\mathcal{B}}}, \mathcal{A}, \mathcal{B})$$

Figure 6.2: NAXOS KAP Protocol. Setup (not in figure), both Alice $\mathcal{A}$ and Bob $\mathcal{B}$ have their own long term secret keys $sk_{\mathcal{A}}$ and $sk_{\mathcal{B}}$ as well as each other's long term public keys $pk_{\mathcal{A}} = g^{sk_{\mathcal{A}}}$ and $pk_{\mathcal{B}} = g^{sk_{\mathcal{B}}}$. Alice begins by creating an ephemeral key $ek_{\mathcal{A}}$. Alice then hashes her long term secret key $sk_{\mathcal{A}}$ and a ephemeral key $ek_{\mathcal{A}}$ to create an ephemeral private key x and calculates ephemeral public key $X = g^x$. Alice sends the ephemeral public key X to Bob. Bob then does the same creating y and sending Y. Alice and Bob have the information necessary to calculate the shared secret $K = K_{\mathcal{A}} = K_{\mathcal{B}}$ [44].

The NAXOS KAP allows Alice and Bob to compute ephemeral private and public keys and can compute a shared key.

$$x = H_1(sk_{\mathcal{A}}, ek_{\mathcal{A}})$$
$$y = H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}})$$
$$K = H_2(g^{x \cdot y}, g^{y \cdot sk_{\mathcal{A}}}, g^{x \cdot sk_{\mathcal{B}}}, \mathcal{A}, \mathcal{B})$$

Each side calculating the same K implies that both sides know their private keys and know the other's public keys. Figure 6.2 shows the complete protocol.

Comparing NAXOS to Diffie Hellman (D-H) [81] the ephemeral keys are x and y. We can consider using the NAXOS construction in standard D-H terminology $x = H_1(ek_{\mathcal{A}}, sk_{\mathcal{A}})$ and $y = H_1(ek_{\mathcal{B}}, sk_{\mathcal{B}})$ since these hashed values result in secret ephemeral random values. Looking at the construction of K,

$$K = H_2(g^{x \cdot y}, g^{y \cdot sk_{\mathcal{A}}}, g^{x \cdot sk_{\mathcal{B}}}, \mathcal{A}, \mathcal{B})$$

Case	Server			Client			Effect
	$sk_{\mathcal{A}}$	$esk_{\mathcal{A}}$	x	$sk_{\mathcal{B}}$	$esk_{\mathcal{B}}$	y	
1	Y	Y	–	–	–	–	Loss of privacy and Authentication
2	–	–	–	Y	Y	–	Loss of privacy and Authentication
3	–	–	Y	–	–	Y	Loss of privacy
4	–	–	–	–	–	–	Secure

Table 6.1: NAXOS security when Keys and intermediate values are known to the adversary. If none of the cases 1-3 are true, the system is secure (case 4).

we find that the first entry in the H_2 calculation is simply $g^{x \cdot y}$, and creates an ephemeral shared value exactly as if it were a normal D-H.

The next two values in the hash H_2 will be the same if and only if the parties know their private keys. The first proves $\mathcal{A}$ knows her private key since $g^{y \cdot sk_{\mathcal{A}}} = Y^{sk_{\mathcal{A}}}$ and the second proves that $\mathcal{B}$ knows his private key since $g^{x \cdot sk_{\mathcal{B}}} = X^{sk_{\mathcal{B}}}$ The final two values $\mathcal{A}$ and $\mathcal{B}$ are the unique identities given to Alice and Bob expressed as a binary string.

The proof in the original paper is that the system is secure unless the attacker has access to $sk_{\mathcal{A}}$ and $ek_{\mathcal{A}}$ or the attacker has access to $sk_{\mathcal{B}}$ and $ek_{\mathcal{B}}$. We can extend this knowledge to include the intermediate value x and $sk_{\mathcal{A}}$ or y and $sk_{\mathcal{B}}$. If the attacker only knows X, to get both $sk_{\mathcal{A}}$ and $ek_{\mathcal{A}}$ he must be able to calculate the discrete log of X to get x and then the preimage of x to get $sk_{\mathcal{A}}$ and $ek_{\mathcal{A}}$, both of which are computationally infeasible. If the attacker knows both intermediate values x and y, the security of K is lost because the attacker is assumed to know Y, $pk_{\mathcal{A}}$ and $pk_{\mathcal{B}}$.

$$K \leftarrow H_2(Y^x, pk_{\mathcal{A}}^y, pk_{\mathcal{B}}^x, \mathcal{A}, \mathcal{B})$$

It should be noted that the system is proven to be still secure if both ephemeral keys are known, that is, the attacker knows any combination of $ek_{\mathcal{A}}$, $ek_{\mathcal{B}}$, x or y except for the combination x and y. Knowing these values, maybe because they may be stick-at values, means that the connection loses PFS but is still secure unless the corresponding long-term key is compromised.

NAXOS++ KAP Protocol

$$\begin{array}{lcl}
\mathcal{A} & & \mathcal{B} \\
ek_{\mathcal{A}} \leftarrow \{0,1\}^{\lambda} & & \\
x \overset{\$}{\leftarrow} H_1(sk_{\mathcal{A}}, ek_{\mathcal{A}}) & & \\
 & \xrightarrow{\quad X = g^{x} \quad} & \\
 & & ek_{\mathcal{B}} \leftarrow \{0,1\}^{\lambda} \\
 & & y \overset{\$}{\leftarrow} H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}}, X) \\
 & \xleftarrow{\quad Y = g^{y} \quad} & \\
 & \xleftarrow{\quad C_{\mathcal{B}} = e_{K_{\mathcal{B}}}(X) \quad} & \\
 & \xrightarrow{\quad C_{\mathcal{A}} = e_{K_{\mathcal{A}}}(Y) \quad} &
\end{array}$$

Figure 6.3: A version of NAXOS that checks that the key is identical on both sides and mitigates for a client with low entropy RNG

Figure 6.1 visualizes the result of the proof in that the system remains secure (case 4) unless cases 1-3 happen. Once K has been calculated, the ephemeral keys $ek_{\mathcal{A}}$ and $ek_{\mathcal{B}}$ as well as the intermediate values x and y must be destroyed.

6.3 NAXOS++

We believe NAXOS needs some extensions if it is to be used as a replacement for TLS. NAXOS needs to confirm that the key exchange was successful and allow clients with flawed RNG to hide that fact from organizations monitoring the Internet.

NAXOS protocol does not check that K is valid

In the mathematical sense, not checking that K was calculated the same on both sides is not a flaw. When $\mathcal{A}$ sends her traffic, she knows that the only person that can decrypt this message is $\mathcal{B}$, but in a practical sense, the key agreement should succeed or $\perp$. Since the entire goal of a KAP is to agree on a key, demonstrating that both sides have agreed on the same key proves that the protocol was successful.

If Alice or Bob do not know their private key, they will not calculate the same K. Without a check, when $K_{\mathcal{A}} \neq K_{\mathcal{B}}$, the only symptom that Alice or Bob will see will be data integrity or decryption failures. Both Alice and Bob need to know that the other knows K, so a simple authenticated encryption of any message will suffice. We propose sending the received public ephemeral key (Figure 6.3), but any message that demonstrates knowledge of K is sufficient. We suggest that a deterministic Authenticated Encryption (AE) or deterministic Authenticated Encryption with Associated Data (AEAD) algorithms (see Section 2.2) since eliminating the use of random nonces is one of the main goals of this work.

Adding a check using the same AEAD algorithm used for the remainder of the connection does not change the protections provided by the KAP and AEAD; it only makes the operation of NAXOS++ clearer to ensure that the KAP was successful.

NAXOS Assumes perfect random numbers

Again, the assumption that random numbers are perfectly random is not a flaw in the mathematical sense. Proofs of security (NAXOS included) assume that $\{0,1\}^{\lambda}$ produces unbiased random numbers. If this random number is stuck, X will be constant for any device with the same public key. If $\{0,1\}^{\lambda}$ has a period of 2^{32}, then X will also have a period of 2^{32}.

Our research has shown that the server-side of a conversation typically can produce reliable random numbers. Servers are also more routinely tested by Internet scans and have a dedicated team focused on reliability and security. The fact that servers are more tested leads us to conclude that we can leverage the server entropy to mitigate the possibility of a low entropy RNG in the client.

Our solution is for the client ($\mathcal{B}$) to add the server entropy to their ephemeral secret key. We calculate the ephemeral private key y as the hash of the long term private key $sk_{\mathcal{B}}$, the ephemeral key $ek_{\mathcal{B}}$ and the received X.

$$y = H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}}, X)$$

Hashing up the values ensures that adding the additional entropy sources to y does not reduce the entropy of y. For a given client ($sk_{\mathcal{B}}$) the entropy of Y will be at least X (if the client RNG has no entropy) and the entropy of X plus λ if the RNG has perfect entropy (here $S(X)$ is the entropy in X):

$$S(X) \leq S(Y) \leq S(X) + \lambda$$

We also note that low entropy $ek_{\mathcal{B}}$ that does not repeat will never be detectable for two reasons. First, it is hashed up, and second, the value in the messages will be the ephemeral public key $Y = g^y$. Repeated values are mitigated by the addition of the entropy of X, but repeated Y are still possible and discussed in Section 6.5.

Long-term public keys have also been a problem [8], so it is recommended that client public keys are installed as a manufacturing step instead of a first initialization step. Installing the client certificates at manufacturing allows for better public keys and the ability of the manufacturers to certify these keys.

6.3.1 NAXOS++ Security Proof

The complete NAXOS++ protocol is shown in Figure 6.3. There are two differences between NAXOS and NAXOS++, the inclusion of X in the calculation of y, and the exchange of messages to show that K is valid.

NAXOS++ security is not affected by the inclusion of X in y

We start by simplifying the discussion by using the intermediate variables x and y. The intermediate variables are:

$$x = H_1(sk_{\mathcal{A}}, ek_{\mathcal{A}})$$
$$y = H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}}, X)$$

Note that the only difference in the NAXOS++ protocol is the inclusion of X in the calculation of y. The attacker needs to recover K. To do that, they need to perform the following calculation choosing one value from each column. The attacker is assumed to know g, X, Y, $pk_{\mathcal{A}}$, $pk_{\mathcal{B}}$, $\mathcal{A}$ and $\mathcal{B}$ the what the attacker does not know is in red.

$$K = H_2\left(\begin{array}{c} X^{y} \\ Y^{x} \\ g^{x \cdot y} \end{array} , \begin{array}{c} {pk_{\mathcal{A}}}^{y} \\ Y^{sk_{\mathcal{A}}} \\ g^{sk_{\mathcal{A}} \cdot y} \end{array} , \begin{array}{c} X^{sk_{\mathcal{B}}} \\ {pk_{\mathcal{B}}}^{x} \\ g^{x \cdot sk_{\mathcal{B}}} \end{array} , \mathcal{A} , \mathcal{B} \right)$$

As stated above, the attacker does not know both x and y since knowing both breaks the system, but the attacker must know either x or y. If the attacker knows x or y, to recover K they must also know $sk_{\mathcal{A}}$ or $sk_{\mathcal{B}}$ respectively.

The attacker knowing $sk_{\mathcal{A}}$ or $sk_{\mathcal{B}}$ can only be because the key is compromised or calculated. Calculating the secret key from the public key reduces to the Computational Diffie Hellman problem (CDH).

The attacker knowing x or y can only be because x or y are compromised or calculated. There are three ways to calculate x or y, either by solving CDH and reversing X or Y, attacking the hash function or by recovering both the secret and the ephemeral keys. Attacking the hash function is not possible because of the assumed collision resistance of the hash function, which implies that the hash function is a secure pseudorandom function that outputs sufficient entropy. Recovering x or y is possible to guess the ephemeral and secret keys. Even if we assume we know the secret key, guessing the ephemeral key is still difficult. The only difference between NAXOS and NAXOS++ is the inclusion of a known variable X in the hash function for y. This difference does not change the arguments above.

Assuming the intermediate values x and y are not compromised, the CDH problem is not solved and hash function is collision resistant, the only way to discover the key K are to recover either:

- $sk_{\mathcal{A}}$ and $ek_{\mathcal{A}}$
- $sk_{\mathcal{B}}$ and $ek_{\mathcal{B}}$

which is the same bound as the NAXOS $\square$

NAXOS++ security is not affected the additional checks

The use of the value K does not affect the proof.

NAXOS–TLS KAP Protocol

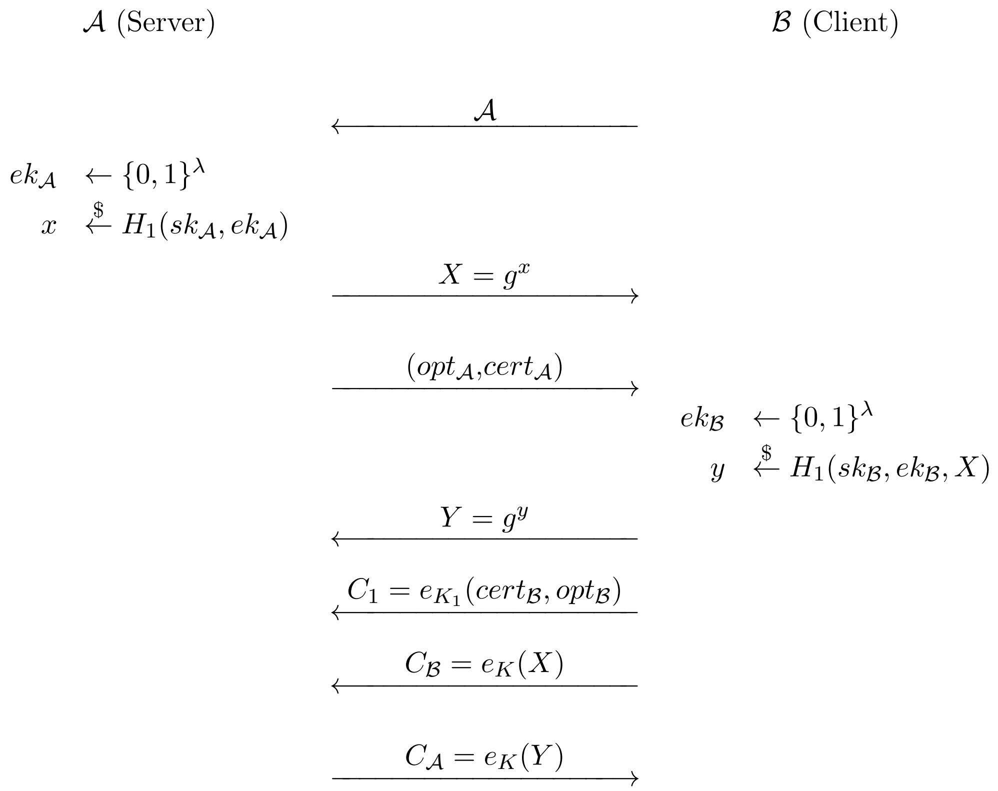
$\mathcal{A}$ (Server)
$\mathcal{B}$ (Client)
$\mathcal{A}$
$ek_{\mathcal{A}} \leftarrow \{0,1\}^{\lambda}$
$x \overset{\$}{\leftarrow} H_1(sk_{\mathcal{A}}, ek_{\mathcal{A}})$
$X = g^x$
$(opt_{\mathcal{A}}, cert_{\mathcal{A}})$
$ek_{\mathcal{B}} \leftarrow \{0,1\}^{\lambda}$
$y \overset{\$}{\leftarrow} H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}}, X)$
$Y = g^y$
$C_1 = e_{K_1}(cert_{\mathcal{B}}, opt_{\mathcal{B}})$
$C_{\mathcal{B}} = e_K(X)$
$C_{\mathcal{A}} = e_K(Y)$

Figure 6.4: NAXOS–TLS protocol extends the NAXOS++ protocol to announce the certificate the client wants to connect to, exchange certificates, and options. The Client certificate and client TLS options are transferred confidentiality so that only $\mathcal{A}$ can know who is connecting, and passive monitoring can not determine the implementation.

6.4 NAXOS–TLS: A TLS AKE protocol with client certificate

Enhancing the NAXOS++ AKE protocol for use by TLS requires extension in two ways. The first is where both the client and servers do not have each other's certificates a priori. The second is that the client certificate and options are passed to the server encrypted so that passive monitoring cannot fingerprint the implementation. Only the server can determine who the client is. Section 6.5 discusses how the NAXOS–TLS protocol reduces the vulnerabilities that can be determined by passive monitoring while continuing to allow active tests to be able to discover these problems.

The agreed key K is unchanged from NAXOS++. We introduce a new key K_1, which is only used for Bob to communicate his certificate and TLS options privately to Alice before the agreed key can be calculated. Before K_1 is used Bob must validate $cert_{\mathcal{A}}$ to get $pk_{\mathcal{A}}$ and $\mathcal{A}$ as well as presenting a MITM from getting $opt_{\mathcal{B}}$

$$K_1 \quad = H_2(X^y, pk_{\mathcal{A}}^y, \mathcal{A})$$

K_1 is the same as K with the authentication of Bob removed since without Bob's certificate, Alice does not know $pk_{\mathcal{B}}$ or $\mathcal{B}$. Traffic from $\mathcal{B} \rightarrow \mathcal{A}$ using key K_1 is secure since Bob knows that the only person that can decrypt the message knows Alice's long term private key. The messages from $\mathcal{B} \rightarrow \mathcal{A}$ using key K_1 also enjoy the same proof of security and PFS as key K.

The Figure 6.4 describes the complete protocol. The protocol starts with Bob sending the certificate's name that they want Alice to return. Note that this enables cloud infrastructure to host hundreds of services with one certificate and hide which exact service is desired in the encrypted TLS options. Encrypting the SNI has been discussed in the IETF [40]. Google has a service, "SSL Proxy Load Balancing" [83] that could take advantage of the additional privacy.

Bob sending a specific certificate $\mathcal{A}$ replaces the SNI option in the Client Hello message. The server $\mathcal{A}$ generates the ephemeral private key $ek_{\mathcal{A}}$ and then replies with the ephemeral public key X. Alice's TLS options $opt_{\mathcal{A}}$, and certificate $cert_{\mathcal{A}}$ are sent in the clear. The server certificate and server options are not encrypted because they are available to anyone who connects and asks for them.

Bob then validates Alice's $cert_{\mathcal{A}}$ and whether Bob is compatible with the value of $opt_{\mathcal{A}}$. If the validation fails, Bob disconnects. Simply disconnecting is valid for two reasons: Sending an error to the server gives away much information about the client. There is nothing the server can do to remedy the situation. If the validation succeeds, the certificate provides $pk_{\mathcal{A}}$ and her identity $\mathcal{A}$.

Bob calculates the ephemeral key $ek_{\mathcal{B}}$ and the ephemeral private key x. Bob can now calculate both keys K, K_1 and the messages C_1, $C_{\mathcal{B}}$. The encryption function e_k is required to be a deterministic authenticated encryption algorithm such as AES GCM SIV [38]. Using an authenticated encryption algorithm that leaks raw random values defeats the purpose of NAXOS–TLS. Bob then replies with Y, C_1, and $C_{\mathcal{B}}$.

Alice receives the messages from Bob. Alice does not have any authentication of Bob at this time, only a shared K_1 with *someone*. Alice needs to decrypt C_1 and validate $cert_{\mathcal{B}}$ and $opt_{\mathcal{B}}$ before proceeding. If the certificate is not valid or the options are incompatible, Alice can send an error to Bob and disconnect. The validated certificate provides $pk_{\mathcal{B}}$ and Bob's identity $\mathcal{B}$. Bob's identity is validated by decrypting and checking $C_{\mathcal{B}}$ which proves that bob knows K which can only be calculated if Bob knows $ek_{\mathcal{B}}$, and $sk_{\mathcal{B}}$.

Once K has been calculated, the ephemeral keys $ek_{\mathcal{A}}$, $ek_{\mathcal{B}}$, x, y and K_1 must be securely erased.

The security proof of NAXOS–TLS is unchanged from the original NAXOS. The usage of NAXOS–TLS without a certificate, in a Peer to Peer (P2P) environment, and a post-quantum secure version are left to Future Work (Chapter 7).

NAXOS–TLS hides the client certificate and options from passive Internet monitoring and ensures that clients have security even if they do not provide a secure random number generator. Implementations with a secure random number generator have additional security benefits.

6.4.1 NAXOS–TLS Security Proof

The complete NAXOS–TLS protocol is shown in Figure 6.4. The only difference in the proof of NAXOS–TLS from NAXOS++ is the use of K_1 to send information from $\mathcal{B}$ to $\mathcal{A}$. Unless the K_1 is compromised or calculated it is guaranteed that only $\mathcal{A}$ can read the message C_1 from $\mathcal{B}$ because calculating K_1 requires $sk_{\mathcal{A}}$

We use the same argument as was used for NAXOS++. The intermediate variables are the same:

$$x = H_1(sk_{\mathcal{A}}, ek_{\mathcal{A}})$$
$$y = H_1(sk_{\mathcal{B}}, ek_{\mathcal{B}}, X)$$

The attacker needs to recover K_1, and to do that, they need to perform the following calculation choosing one value from each column. The attacker is assumed to know g, X, Y, $pk_{\mathcal{A}}$, and $\mathcal{A}$. What the attacker does not know is in red.

$$K_1 = H_2(\begin{array}{c} X^{y} \\ Y^{x} \\ g^{x \cdot y} \end{array} , \begin{array}{c} pk_{\mathcal{A}}{}^{y} \\ Y^{sk_{\mathcal{A}}} \\ g^{sk_{\mathcal{A}} \cdot y} \end{array} , \mathcal{A} , \mathcal{B})$$

As stated above, the attacker does not know both x and y since knowing them breaks the system, but the attacker must know either x or y. If the attacker knows x, to recover K_1, they must also know $sk_{\mathcal{A}}$. If the attacker knows y, K_1 is compromised.

The attacker knowing $sk_{\mathcal{A}}$ can only be because the key is compromised or calculated. Calculating the secret key from the public key reduces to the CDH.

The attacker knowing x or y can only be because x or y are compromised or calculated. There are three ways to calculate x or y, either by solving CDH and reversing X or Y, attacking the hash function or by recovering both the secret and the ephemeral keys. Attacking the hash function is not possible because of the assumed collision resistance of the hash function, which implies that the hash function is a secure pseudorandom function that outputs sufficient entropy. Recovering x or y is possible to guess the ephemeral and secret keys. Even if we assume we know the secret key for either party, guessing the ephemeral key is still difficult because the proof assumes perfect entropy. See Section 6.5 for a discussion of the effects of less than perfect entropy.

Assuming the intermediate value x is not compromised, the CDH problem is not solved and hash function is collision resistant, the only way to discover the key K_1 are to recover either:

- $sk_{\mathcal{A}}$ and $ek_{\mathcal{A}}$
- $sk_{\mathcal{B}}$ and $ek_{\mathcal{B}}$

which is the same bound as the NAXOS. □

6.5 Residual Vulnerabilities

The NAXOS–TLS system retains the original security proof, which allows "an adversary to reveal any subset of the four which does not contain both the long-term and ephemeral secrets of one of the parties" while remaining private and authentic. NAXOS–TLS protects the client's identity and reduces the vulnerabilities uncovered by passive monitoring while allowing active tests to discover these problems.

The only random number related residual vulnerability that we need to worry about are numbers that repeat. Low entropy random numbers that do *not* repeat are not an issue with NAXOS–TLS because the Decisional Diffie Hellman assumption (DDH) [72] tells us that if we get two X values, say X_1 and X_2, and $X_1 \neq X_2$ then all we learn is that $x_1 \neq x_2$. Even if private values x_1 and x_2 differ only by one bit, the public values are indistinguishable. The fields available for the attacker are the $pk_{\mathcal{A}}$, $pk_{\mathcal{B}}$, X and Y. All of these values are Diffie-Hellman public values.

6.5.1 Passive monitoring

Servers are unchanged from today; they will still be actively tested. They are intentionally designed to be scanned and monitored.

Repeating $pk_{\mathcal{A}}$ represents low entropy on the long-term key. This is detectable by checking the certificate $cert_{\mathcal{A}}$ which contains both the $pk_{\mathcal{A}}$ and $\mathcal{A}$. A low entropy $sk_{\mathcal{A}}$ is only visible if there are two machines with the same $pk_{\mathcal{A}}$ but unrelated identities. Sharing a public key indicates that the machines are vulnerable to each other, and discovering why can lead to the recovery of $sk_{\mathcal{A}}$.

Repeating X identifies servers with low entropy ephemeral keys. Finding duplicate X means that the machine is more vulnerable because if the attacker can determine the flaw in the RNG, the machine has lost PFS. Losing PFS means that, if sometime in the future, the long-term key is compromised, past traffic is vulnerable.

NAXOS–TLS is robust to low entropy client keys against passive monitoring. The client $pk_{\mathcal{B}}$ is not available for passive monitoring because it is encrypted by K_1. A low entropy client can only be passively discovered if the server is also low entropy. In this case, both X and Y will be repeated. If both have low entropy, the traffic loses PFS, and if either of the long-term keys is compromised, then the security has failed.

6.6 Active Testing

The changes to the TLS AKAP allow the testing of the implementations. Active testing of TLS Servers continues as today, they are on the Internet, and anyone can connect to them. Actively testing clients will require the clients to "opt-in" to testing by connecting to a machine specifically designed to test clients.

Identifying servers with duplicate long term keys can be accomplished by scanners that detect certificates for two servers with different identities $\mathcal{A}_1 \neq \mathcal{A}_2$ having the duplicate public keys $pk_{\mathcal{A}_1} = pk_{\mathcal{A}_2}$. Actively checking the X to make sure it is not repeating will ensure that the $ek_{\mathcal{A}}$ is not stuck. Testing a million times and not seeing any repeated value is no guarantee that the next test will not see a repeated value, but a test like this is better than nothing.

Actively identifying clients with duplicate long-term keys can be accomplished by clients intentionally connecting to sites that harvest certificates looking for clients with different identities having the duplicate public keys $pk_{\mathcal{B}}$. Actively checking the Y to ensure it is not repeating requires the client to connect to a specific server that will attempt to determine if the client's random number generator is stuck or repeating. To test the client's Y, the server must hold its X constant. The client can then make many connections to the server will ensure that the $ek_{\mathcal{A}}$ is not stuck. Testing a million times and no repeats happen is not guaranteed that the next time will not have a duplicate random number, but a test like this is better than nothing.

Mathematically, NAXOS is secure because of $X = g^{H_1(ek_{\mathcal{A}}, sk_{\mathcal{A}})}$ can not be reversed due to the DDH and the Preimage Resistance assumption. These are assumptions because there is no proof that these assumptions are difficult, just that no one has found a tractable way for these to be solved. Even if these assumptions are computationally infeasible, if the values $(ek_{\mathcal{A}}, sk_{\mathcal{A}})$ have low enough entropy, and trying them all is feasible, none of the assumptions above apply.

6.7 Intentional Leakage

There is no encryption protocol employing random numbers as the key material that can not be intentionally subverted. If Alice is conversing with Bob, Alice could create a random number a priori, give that value to the adversary Eve, and then use that key in an encryption protocol with Bob.

6.8 Summary

The residual vulnerabilities of NAXOS–TLS related to bad random number generators are limited to repeated values. Internet scans can detect servers with low entropy RNG that repeat. Clients with low entropy RNG where the server does not have a low entropy RNG will not be detected or have an issue.

Neither a wise man nor a brave man lies down on the tracks of history to wait for the train of the future to run over him.

Dwight D. Eisenhower

Chapter 7

Future work

As a part of this work we will be making all the programs available in open source, MIT license without any IP claims. The data used in this thesis is also available to academic institutions for their research.

Differentiators other than repeats

We believe that there are implementations out there without repeating random values that do not have full entropy. To find these implementations we need to find some discriminator that can provide evidence that a group of numbers are not random. To our knowledge there is no methodology to finding these discriminators other than trial and error. It is possible to use tools like Diehard and Diehader [50, 51] if significantly more numbers can be acquired.

It is interesting to note that NAXOS, NAXOS++ and NAXOS–TLS are immune from low entropy random numbers that do not repeat. These protocols make only the ephemeral public keys available and access to raw random numbers has been eliminated.

Self contained module that can be deployed in web servers or in Zeek

Our process requires $\mathcal{O}(n)$ storage. Reducing the storage requirement is necessary if we are going to be widely deployed solution. One solution that we looked at, and discarded for this thesis, was to use Bloom filters that accumulate information and capture potential candidates but can not guarantee that a candidate is real unless the number repeats more than 2 times. The data we have already gathered could be useful for this research.

Why is TLS so fragile?

TLS allowing implementation fingerprinting as well as using random numbers directly in the messages has to be something that people who study bad random number generators has to know is an issue. Given the issues around over the years with Crypto AG [3] and EC-DRBG [19], and the number of government employees and contractors at IETF meetings, were they passively watching a fragile protocol being developed or did they help it along? The developers of SSL and TLS are still around so, similar to Don Coppersmiths' discussion of the development of DES [84], discussions by the developers of TLS and SSL is warranted.

Fingerprinting implementations if client options are encrypted

In the NAXOS–TLS, the options are encrypted and sent with the client. The length of the ciphertext will leak information about the implementation. Fixed length options are possible, but probably resisted by the IETF. Would the length of the options leak the implementation? It might be possible if the server could present a list of modes and the client chooses only one without further customization, but this leaves the length of the certificate to possibly define the implementation.

NAXOS–TLS in Peer to Peer

NAXOS++ can be considered for a Peer to Peer, but the peer that is first (Alice by definition) will not have the same protection Bob has.

NAXOS–TLS as mitigation for SNI in the clear

C. Huitema has written about the need for Server Name Identification (SNI) Encryption in TLS [40] especially in the case where centralized bulk decryption at the edge of a datacenter for 1000s of services in the data center. For example, in Section 4.3.1, there were many hosts being addressed with a suffix of `google.com`. With NAXOS–TLS, the suffix can be sent as $\mathcal{A}$ in the clear and the full $\mathcal{A}$ sent in the encrypted payload.

Traffic Analysis

Even if the traffic is perfectly encrypted, there is a lot to be learned about traffic even if all we know are the source and destination IP addresses. We know that governments *"deliberately reroute Internet communications"* [85] so that it can be monitored. There can be no doubt if Alice is sending packets to `chat.signal.org`, she is engaging in an encrypted communications with someone using the Signal secure messaging application. Timing these messages can also leak the destination. What can be done?

Post quantum

At this time all post quantum algorithms implement a Key Encapsulation Mechanism (KEM) [74] instead of a Key Agreement Protocol (KAP) [82]. This implies that a NAXOS method needs to be significantly changed.

The lack of sophistication of our methods and findings makes it hard for us to believe that what we have presented is new, in particular to agencies and parties that are known for their curiosity in such matters.

Arjen Lenstra

Chapter 8

Conclusion

eCommerce drives the need for secure Internet communications. Applications from shopping sites to business banking would not be possible if we could not trust that the communications are secure. The vast majority of this trust has been placed in Transport Layer Security (TLS) protocol. We know that the TLS protocol is proven secure, but it is critical to know whether or not the implementations of this protocol are also secure.

We have chosen to analyze one facet of the TLS protocol: TLS's use of public random numbers. Many papers have shown that random numbers are difficult to get right. One can assume governments have analyzed these random numbers, but a survey of TLS public random numbers used has never been published.

The results of our analysis contain both good news and bad news. The good news is that we could not find any consumer browsers using low-quality random numbers.

The bad news is that we found less common TLS implementations with inadequate random number generators or constants as their Hello Random values. In a perfect world, we should never see a single duplicate random number, yet we found 29,884 in five months of data. One case was a Google-compatible device that had thousands of repeated Client Hello Random values. The other mistakes seem programmatic, where random number generators are stuck consistently producing the same number. We have verified that at least one of these implementations offers no security. Two implementations put 28 bytes of zeros as their random numbers.

High-quality random numbers are an important assumption about the security of the TLS protocol. Implementations that violate high-quality random numbers assumptions required by the proof of security are vulnerable to attack. Future researchers may need to show how these implementations are vulnerable, but we will focus on why these errors keep happening.

History has shown that "naming and shaming" is inadequate to wake up vendors to fix their products. They may feel that their solution is "good enough" or cannot push fixes to their devices. In any case, the number of vulnerable devices is expected to continue to increase *after* the problems are published.

We believe that Cryptographic algorithm designers need to learn from time-tested information security methodologies and government cryptography practices.

Information security methodologies rely on defense-in-depth. Defense-in-depth assumes that layers of security will be breached, human errors happen, and in general, prepare for the unexpected. Cryptographers assume that algorithms are secure and remain so, assume that mistakes will not happen, and expect their assumptions to be met. A cryptographer's proof of security can be considered fragile since many things can (and do) go wrong with the implementation. In this sense, the attitude of the Cryptographer is dissimilar to the information security professional.

Government cryptography focuses on security as the outcome. They manage the process from the cradle to the grave. Government organizations take responsibility from protocol design to device decommissioning. Government cryptographers also focus on what information is leaked by the protocol and how the adversary may use that information. They realize that the key to increasing the quality of the outcome is to learn from their mistakes and look at the entire process to make sure mistakes do not happen again.

There is no authority with cradle to grave responsibility for the Internet. The IETF controls standardizing protocols but has no authority over the development, testing, products, and operations. When implementation mistakes happen, some cryptographers laugh at the engineers while claiming *their* proof is still correct.

The design of cryptographic algorithms and protocols is based on the language of mathematics, where proofs are valid regardless of how complicated they are. For example, Wiles's proof of Fermat's last theorem is 129 pages long, and regardless of how complex the document is, it is still valid. The beauty of Wiles's proof is that the assumptions are simple and easily stated. History has shown that cryptographers' proofs of security have not served cryptographic users well because, time and time again, implementations are not perfect, and the assumptions are not met.

Cryptographic algorithm and protocol designers also need to consider issues beyond the proof. First and foremost, they must state all the assumptions they are making, including assumptions about all the algorithms they are using. They need to consider the "blast radius" when any of these algorithms are broken and weigh each component's value against the risks that the component enables. They need to understand what information is being given to the adversary and what value that information has.

We have shown that the TLS protocol is vulnerable as implemented and deployed in the real world. Some implementations are not secure, and some implementors seem uninterested in fixing their insecure implementations. We have described a long-term fix for the issues we found. We guide future protocol designers to look beyond the proof and consider their protocol from the perspective of a future where imperfect implementations happen and algorithms fail.

We have described a provable secure replacement TLS protocol that is deterministic in that it had eliminated the Client and Server Hello Random values. This Key Agreement Protocol either works or outputs $\perp$ and is significantly better at hiding both the implementation and the quality of the random numbers.

TLS is incredibly important. The privacy it enables can be the difference between life and death. We must embark on a path to improve TLS for the people who have no choice but to trust the protocol for their security.

The enemy knows the system.

Claude Shannon

Appendix A

True Random Number Generator

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. As has been pointed out several times, there is no such thing as a random number – there are only methods to produce random numbers, and a strict arithmetic process is not one of them." John Von Neumann, 1951 [87].

This chapter describes a true random number generator using CMOS image sensor noise. This methodology takes the opposite of Von Neumann's "strict arithmetic process" by adding additional unknown (random) information that Shannon called Entropy [88].

We can call this random information s for seed, S for all possible seeds, and the entropy present in the set of all seeds in bits $H(S)$. The result of the random number generator combines the s with a strict arithmetic function f such that the resulting data $d = f(s)$. We use the form $|d|$ to denote the integer number of bits in d.

At this point the methodology bifurcates based on the relationship between the entropy $H(S)$ and the length of the data created in bits $|d|$.

$$\text{CSPRNG: } H(S) < |d|$$
$$\text{TRNG: } H(S) \geq |d|$$

Figure A.1: Raspberry Pi HD Camera with the Sony IMX477 sensor (from [86])

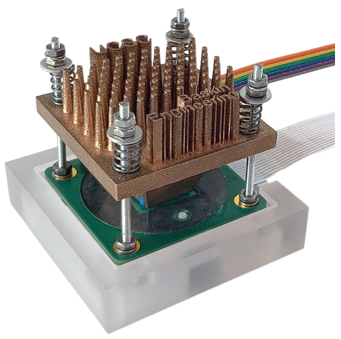
Figure A.2: Temperature control fixture showing heat sink and camera board.

CSPRNG takes a seed s and expands it into a number d where the length of $|d|$ is larger than $H(S)$. This method is conjectured secure because the function f contains a cryptographic algorithm assumed to create indistinguishable data from random. While the number can be indistinguishable from random, from an information-theoretic perspective, the actual amount of entropy in d can never be more than $H(S)$.

Most CSPRNGs get their s from various sources. Some may be from a source that can be traced to random physical properties and may have some sources that are believed to have entropy. These methods have been shown to have issues. We believe that CSPRNG has risks significantly higher than the alternative.

True Random Number Generator (TRNG) is an alternative that creates a random number directly from the seed s, ensuring that the resulting d is an ideal random sequence whose length can be no greater than $H(S)$. TRNGs get their seed s from some source with random characteristics traced back to a known random physical process. If more d is needed, we simply repeat the process. From an information-theoretic sense, as long as there is the expected amount of entropy and the whitening process is correct, all the values of d will be Ideal Random Sequence [89].

Based on this definition, CSPRNGs do not produce Ideal Random Sequences because the length of the stretched data is larger than the entropy. It is only a TRNG that can produce ideal random sequences.

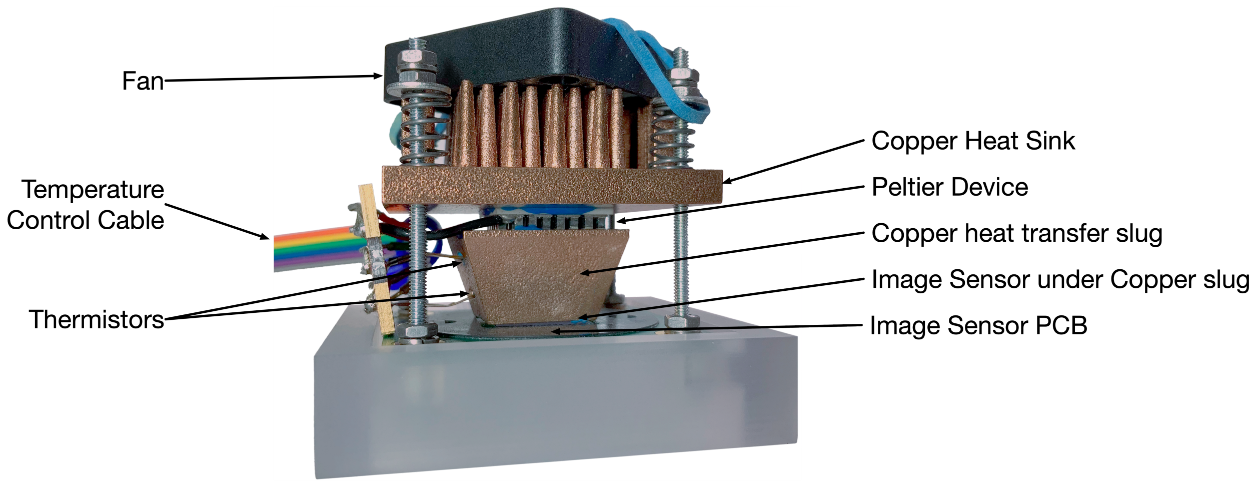
Fan
Temperature Control Cable
Thermistors
Copper Heat Sink
Peltier Device
Copper heat transfer slug
Image Sensor under Copper slug
Image Sensor PCB

Figure A.3: Another perspective of the temperature control fixture showing (top to bottom) the fan, the heat sink, the Peltier device, heat transfer slug with an upper and lower thermistor, the HD Camera PCB with the lens mounting ring removed. Circuitry connecting the Peltier device and thermistors to the ribbon cable is on the left.

A.1 Test Device

We describe a True Random Number Generator from a commodity imaging sensor at room temperature (Figure A.1). We demonstrate this capability using a 12,330,240 pixel Sony IMX477 sensor [90] which is more commonly known as the Raspberry Pi HD Camera. This device and a Raspberry Pi processor together commonly sell for less than $100.

We characterize the effect of heat by stopping the imaging sensor from receiving any light. We use a custom fixture shown in Figure A.2. We attached a heater/chiller to the sensor covering the light-sensitive face to show that temperature alone can affect the image sensor's entropy.

Figure A.3 shows the details of the device starting from the top with a small fan down to a holder for the image sensor. The fan and 3D printed copper heat sink create a stable temperature that the Peltier device can leverage. This Peltier device is rated to be able to create a Δt of 83.2C [91] from the room temperature heat sink. This delta can drive the 3D printed copper slug either above or below the room temperature heat sink based on the polarity of the current applied to the Peltier device.

The polarity and the current to the Peltier device are controlled by a proportional–integral–derivative controller (PID controller) [92] running in a Raspberry Pi, which adjusts both the polarity and current until both of the thermistors register the target temperature. The Raspberry Pi was also used to collect the images from the sensor.

The copper slug is glued to the face of the sensor using thermal paste. The combination of slug and paste ensures that no light is making its way to the device.

We have used this system to drive the temperature of the image sensor from 10C through 50C in 10C increments while acquiring 1000 images at each temperature. The entropy of a system cooled to a temperature approaching absolute zero approaches zero. A better perspective is to consider the measurements from 283 K to 323 K. The results are shown in Section A.5.6 and show that the sensor, devoid of light, is sensitive to temperature and that there is sufficient entropy throughout this range.

A.2 Cryptographically Secure Pseudo-Random Number Generator

Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) "stretches" the seed s in an assumed cryptographically secure way to generate a larger number of pseudo-random bits. This method has been codified in papers too numerous to mention, books [93, 94] and standards [89]. Most random number generators in use today utilize several sources of data that are presumed to include entropy, and that entropy is then stretched to provide the random numbers needed for an application.

A.2.1 Intel Digital Random Number Generator

Intel Ivy Bridge CPUs have a random number generator with the typical CSPRNG functionality of expanding the seed to produce the data that is consumed by the application [95, 96]. The "normal" way that the Intel Digital Random Number Generator is used is as an additional source of entropy to the Linux CSPRNG.

A.2.2 Linux CSPRNG

The Linux CSPRNG is arguably the most widely used. Two of the methods do not block even if there is not enough entropy in the system, `/dev/urandom` and `getrandom()`. Both of these methods rely on the ChaCha20 algorithm [97] as they stretch the `crng_state` to produce random numbers.

The biggest failures of CSPRNGs, including the Linux CSPRNG, include:

- CSPRNG failures caused by providing random numbers before enough entropy has been accumulated [8]. This failure is common because a device as manufactured, or VM on first boot, is always the same and the first thing a device like this will do is create long-term keys. If the random number generator is allowed to provide random numbers too soon, multiple devices may use the same s and generate the same random numbers with a probability far higher than expected. This failure mode is still common today [9].
- CSPRNG failures caused by not collecting enough entropy because of improper assumptions about the amount of entropy in the data coming into the system.
- CSPRNG failures caused by running the system out of entropy. The Linux output `/dev/urandom` can extract more entropy than is in the system. This has been changed recently to add ChaCha20 encryption so that `/dev/urandom` is at least as secure as ChaCha20 [98].
- CSPRNG failures caused by assuming that the system is secure and that the submitters are non-malicious [99].
- Assuming supply chain attacks have not occurred [100].
- Algorithms fail. The Linux CSPRNG relies on SHA-1, which has shown to have collisions [101], but the usage here, while maybe secure "enough" for this usage, has not been proven. ChaCha20 is thought to be secure at this time.

The Linux CSPRNG is by most measures a complicated piece of software.

"complexity is the worst enemy of security" Bruce Schneier, 2012 [102].

The Linux CSPRNG may indeed be secure, but it may be "overkill" for many applications. It is helpful for applications that are already running on Linux machines, but the non-blocking random APIs should be avoided. On scaled-down Linux machines, care needs to be taken to ensure that they have the entropy they need.

A.3 True Random Number Generators

True Random Number Generators (TRNG) are fundamentally different and significantly less complicated. A TRNG has only two steps, accumulate a known amount entropy from an understood source of randomness and then whiten the data. Whitening is a technique to eliminate bias and other non-random artifacts from raw random numbers and produce ideal random sequences with 1 bit of entropy for each bit output.

Because the TRNG is not drawing from a constant entropy pool, each output of a TRNG is a block of random numbers that are unrelated to any other previous or subsequent blocks. Because the blocks are independent, a TRNG does not require the same complicated buffering of entropy estimations.

If we can quickly and reliably create random numbers from an inexpensive source with far more entropy than we need, why bother with the complicated system of CSPRNGs or the potential that their cryptographic functions are broken in the future?

A.3.1 Existing TRNG

There have been many true random number generators both in theory and in practice [103]. Examples of TRNG include:

- Rolling physical dice in a machine [104]. The device mechanically rolls dice and then images the position they end up. This machine has the advantage of seeing and understanding the randomness but has the drawback that the machine is large and expensive considering the rate at which random numbers are produced.
- Measuring the noise from a single avalanche diode[105]. The noise is a well-understood phenomenon.
- Taking hundreds of free running oscillators and mixes them down to a single bit [106].
- Taking a beam of photons to a beam splitter and counting which path the photons take [107].
- Placing a Cesium-137 radiation source in front of a radiation monitor[108].

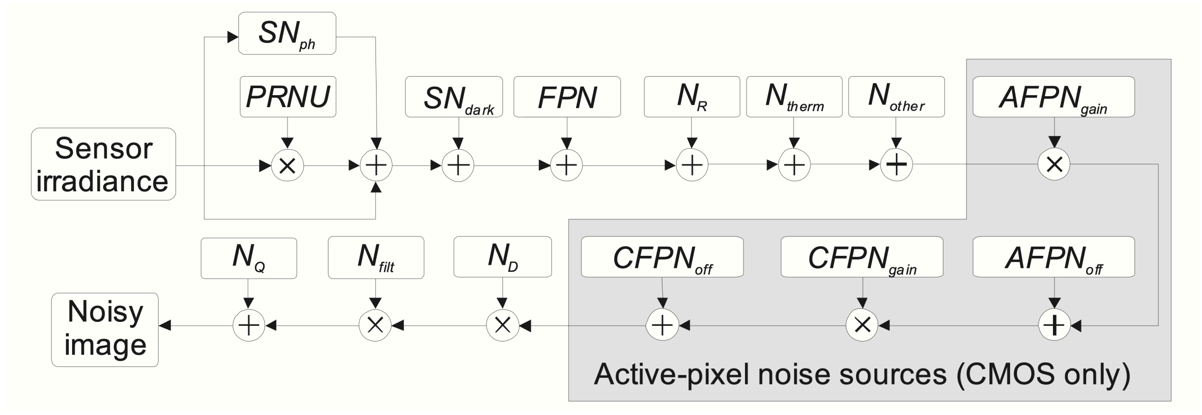

Figure A.4: "A diagram of the noise-flow for a typical camera. The noise sources in the shaded area represent additional noise sources present in CMOS cameras only. Plus symbols refer to additive noises, whereas crosses represent multiplicative noise sources" [110].

- Getting random information from CMOS image sensors includes calculating the entropy from a light source [109]. This method has several similarities to this paper but requires a calibrated light source. We show that the thermal background noise is enough to produce high-quality random numbers without the need for an external light source.

A.4 CMOS Sensor as a Random Source

Virtually every phone today contains at least one CMOS Sensor with tens of millions of pixels. Each one of these pixels is a reliable source of noise (entropy). We leverage the noise to create both high-quality TRNG.

Noise in CMOS sensors has been widely studied [111, 112] in order to understand and attempt to reduce it. Efforts to eliminate noise concentrate on cooling, but nothing has successfully eliminated noise at room temperature. We embrace this noise.

This noise is present in the device-independent of the image being captured and is present when the camera is covered or not. Figure A.4 shows the various components that add illumination independent noise to the image. “Illumination independent noise sources are present at every pixel regardless of whether they are being illuminated or not. When combined, they provide a minimum level of noise present in the captured image.” [110]. We only assume that these noise sources exist and can be measured.

Pixel noise of specific sensors is measured and characterized as μ_d electrons/second at room temperature[113]. Sensors used in photography where low noise in the image is desirable should select a lower μ_d. A higher μ_d is better and more common in consumer-grade inexpensive CMOS sensors as a source of entropy. In any case, the selection is not critical to the discussion; all we need to know that $\mu_d > 0$. Each pixel location has its own μ_d because of manufacturing non-uniformity of the array itself. Additionally, it is known that the accumulations of electrons in the conduction band over time are based on Poisson distribution. μ_d also increases exponentially with the temperature [114].

Capturing this noise and creating uniform random numbers can be difficult since the system is designed to hide this noise when taking images. We begin using the CMOS imaging array (or any CCD array) operated in “raw mode”. Raw mode is essential for three reasons:

1. It disables lossy encoding like JPG or PNG since these algorithms intentionally ignore random artifacts that do not affect human perception of the image. These encodings destroy entropy. Disabling JPG and PNG also provides access to the low-order bits.
2. It provides control over the analog to the digital portion of the process, which is important to tune the process for maximum entropy. Most sensors have automatic controls that assume the image is something a human would want to see. These automatic adjustments could destroy entropy.

A.4.1 Calculating Shannon Entropy

To determine how to use the imaging sensor and adjust the whitening algorithm's parameters, we need to calculate the Shannon Entropy of the bits from a "raw" image. Shannon Entropy is a calculation performed on a block of data and calculates the minimum number of bits necessary to represent this data if all the redundancy in the data were eliminated. Shannon Entropy can be thought of as the limit of lossless compression and randomness in a sample. We calculate the entropy in the usual way using the probability of any given pixel value x_i.

Shannon entropy is defined as:

$$\mathrm{H}(X) = -\sum_{i=1}^{n} \Pr[x_i] \log_2 \Pr[x_i] \tag{A.1}$$

where X is the probability distribution of each of the values. For instance, if we assume values of $\{64, 64, 128, 192, 256\}$ we would have a probability distribution of $X = \{\frac{2}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\}$ and the Shannon entropy of those values is $\mathrm{H}(X) = 1.9219$ bits.

A.4.2 Whitening

Whitening is "distilling" the entropy from a source containing entropy, bias, and redundancy. The output of a whitening function is an "ideal random sequence" where:

> "Each bit of an ideal random sequence is unpredictable and unbiased, with a value that is independent of the values of the other bits in the sequence. Before the observation of the sequence, the value of each bit is equally likely to be 0 or 1, and the probability that a particular bit will have a particular value is unaffected by knowledge of the values of any or all of the other bits. An ideal random sequence of n bits contains n bits of entropy." [89].

Ideal random sequences have one bit of entropy for each bit output which means that each bit needs to be

$$\Pr[1] = \frac{1}{2}, \text{ exactly.}$$

The probability of a 1 or 0 is "unbiased", and thus a collection of all possible n bit values $x_i = \{0, 1\}^n$ will be similarly unbiased each having the uniform probability of

$$\Pr[x_i] = 2^{-n}$$

Starting with Shannon's definition:

$$\begin{aligned}
\mathrm{H}(X) &= -\sum_{i=1}^{2^n} \Pr[x_i] \log_2 \Pr[x_i] \\
&= -\sum_{i=1}^{2^n} 2^{-n} \log_2 2^{-n} \\
&= -\log_2 2^{-n} \\
&= n
\end{aligned}$$

For any n bits taken from the output of the whitening algorithm, n bits "ideal random sequence" are produced. The algorithm for whitening can be a hash or other Pseudo-Random Permutation that diffuses without destroying the entropy.

Some analysis of the randomness of data focuses on Min-entropy [89]. Min-entropy is an alternative to Shannon Entropy

$$H(X)_\infty = \max_{i=1}^{n}(-\log_2 \Pr[x_i])$$

which "measures the probability that an attacker can guess the state with a single guess"[96]. Guessing the state is impossible if the whitening step is a one-way function or calculating the pre-image is computationally infeasible. The min-entropy after whitening equals the Shannon entropy after whitening. If f is the whitening function, $H(f(s))_\infty = H(f(s))$.

A.5 From CMOS Sensor to Random Bits

Building a reliable random number generator is not without controversy. When a random number generator is being used in life-critical (medical, military) or safety-critical (Nuclear power, Aircraft), there is a critical need to need to make sure that these numbers are carefully created and genuinely random.

What about TLS, VPN, IoT, or other technologies that are meant to preserve personal privacy? Is there a line that can be set where carefully creating random numbers is not justified? We claim that the line does not exist because we do not know what information will be valuable to some future adversary. Creating a high-quality random number generator that is "good enough" for life-critical or safety-critical is easy. There is no reason not to use a high-quality random number generator for "low risk" applications.

The process of taking data from a CMOS sensor to a collection of random bits is comprised of a series of steps:

1. Check that the operating environment of the system is within norms.
2. Characterize the sensor so that we know the amount of Shannon Entropy the device provides. Characterization of the sensor has to be done at device initialization and periodically over time.
3. Gather raw entropy from the CMOS Sensor.
4. Analyze the raw data to ensure that the entropy source has not failed by checking that the distribution values are within the expected tolerances.
5. Whiten the raw data. The process of Whitening can be described informally as ensuring that result will have 100% entropy per bit (every bit has a uniform probability of being a 1 or 0).
6. Checks that the calculations were performed correctly.

A.5.1 Operational Environment

The most significant environmental parameter for us is temperature. As the sensor is cooled, the amount of noise (entropy) goes down. If the temperature were to approach 0 K, the entropy would approach 0, and the system would not produce random numbers.

The NIST 140-2 document with "Security Requirements for Cryptographic Modules" discusses that at their Security Level 4 against "compromise due to environmental conditions or fluctuations outside of the module's normal operating ranges for voltage and temperature"[115]. Most computer chips and image sensors already have dozens of temperature sensors to ensure that the system operates within normal limits.

If physical tampering is a credible threat, then tamper responsive behavior should be employed [116]. Additionally, other environmental sensors can be used to detect unusual electromagnetic fields that could affect the system's operation [117].

A.5.2 Characterization of the sensor

The technology underlying a CMOS pixel is an analog device that captures entropy in the form of per-pixel background noise. The image sensor's A2D digitizes this analog noise to create random numbers (X). We can calculate the actual $H(X)$ of the data and use that information to ensure we are putting enough pixels into the whitening algorithm.

CMOS Image sensors have several adjustments, including gain, image rate, and others, which affect the captured digital values. The amplifier's gain is set, so pixel noise to a level above "0" and not so high as to become saturated. The base gain and ISO settings result in an average value pixel value of 256.3 [118].

For the device's calibration, a representative number of images need to be taken. This number of images needs can be tuned to the assurance desired, but since it is only required at startup, this does not affect the system throughout. If the expected entropy is not present, the system is not allowed to be operated.

To ensure the device is continuing to function, routine recalibration is needed. Routine recalibration would be similar to drug lot testing, where a small portion of a lot is subjected to additional testing to make sure the lot is acceptable. This routine calculation needs to compare against the expected initial distribution, not the most recent, to protect against a gradual entropy reduction. The rate of recalibration is also related to the level of assurance desired.

A.5.3 Gather Raw Entropy

Gathering the raw entropy involves ensuring that the device is in raw mode while extracting images from the device. The typical Raspberry Pi camera software takes an image and then stores it as both JPG and raw concatenated together. The raw data is the last 18,711,040 bytes in the image. The raw image is in a sensor-specific format where two 12-bit pixels are stored in three bytes. Inside the raw image, these pixels are stored in bands surrounded by additional metadata. We process the image by throwing away the JPG, additional metadata and converting the pixels into an integer array of 16-bit values. We analyze raw images only after making sure that anything other than a pixel is ignored.

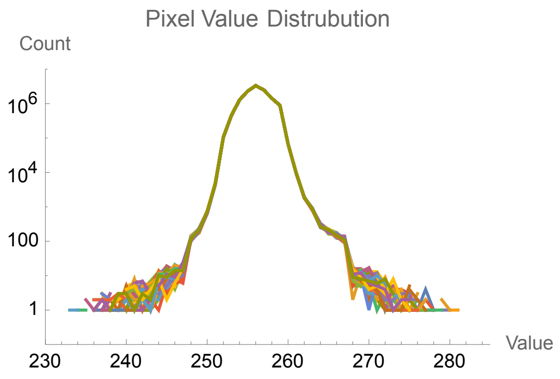

Figure A.5: Distribution of pixel values in 100 images.

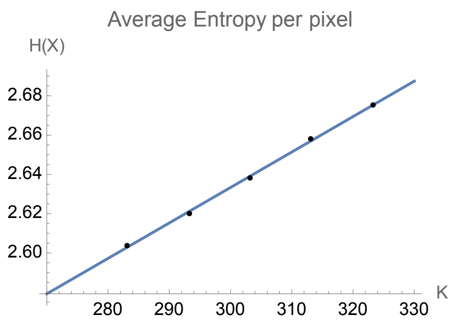

Figure A.6: Average entropy for a single image measured at various temperatures (K) showing the increased entropy with higher temperatures.

There are markers in the metadata of the raw image to make sure what we are seeing aligns with the expected format.

A.5.4 Raw Data Analysis

Once the pixel values are isolated, we need to ensure that every image has a reasonable amount of entropy to detect images that are not providing the expected amount of entropy. From the luck of the draw, a perfectly random image could have lower entropy than average in a way that is indistinguishable from a device failure. To solve this issue, we need to test the entropy of the image and discard any image, potentially even good images, when a single image does not have the entropy needed.

Figure A.5 shows the distribution of pixel values for 100 images at 50. Each color represents a particular image with values in the y axis and the x axis. The values are clustered around 256, which would be far more exaggerated if the y axis were not log-scale.

We get an indication of the quality of the image by running a χ^2 against the expected distribution. If an image does not meet the qualification, it should be discarded. If a large percentage does not meet the qualification, the system should be recalibrated.

A.5.5 Whitening

There are many methods available for whitening [119]. We choose SHA3-512 while making sure that we use it in a whitening mode. SHA-3 is a member of a family of hash functions known as Sponge Construction [120] which absorbs information and squeezes out the hash. We have chosen SHA-3 512 primarily because:

- SHA-3 is a well-analyzed hash function.
- SHA-3 has a large internal state.
- SHA-3's 512-bit output is indistinguishable from random [120] when sufficient entropy is provided.

Shannon teaches that if the set X only contains two values (regardless of the number of bits in these values), the output of any function whose range is X can only have only two values and thus can have at most one bit of entropy. Stated differently, $H(X) \leq H(f(X))$ for any function f and SHA3 512 is no different. If we want full entropy on the 512 bit output, we must put at least full entropy $H(X) \geq 512$ in.

Algorithms like MD-5 and SHA-1 use a Merkel Damgård construction and have an internal state size of 128 and 160 bits, respectively. These algorithms compress the entropy to this internal buffer size, so no matter how much data is put in, only one internal state of entropy can be removed. If these were not broken, they could only process one "state size" chunk at a time.

SHA-3 has an internal state of 1,600 bits and an output size of 512 bits. The total number of bits that can be processed is the smaller of these two numbers. Once we have placed 512 bits of entropy into the buffer, we can squeeze 512 bits out. Using data from the Results Section A.5.6 the 20 sensor has 2.61995 bits of entropy in every pixel. For 512 bits of entropy, we need to absorb 195 pixels to squeeze out 64 bytes of ideal random sequence.

A.5.6 Calculations Check

All of the calculations performed above must be completed at least twice because computers can fail intermittently, making mistakes that can lead to data integrity failures [121]. The assumption here is that a random computational mistake will probably not happen in the same way the second time.

A.6 Results

One thousand images were captured at temperatures from 10C (or 283 K) to 50C at (or 323 K) 10C steps. To eliminate the per pixel manufacturing variance, we averaged the entropy of each pixel over the 1000 images. Figure A.6 shows that at 283 K there was $H(X) = 2.6041$ random bits per pixel and at 323 K there are $H(X) = 2.67542$ bits. Using 293 K as room temperature, the sensor will produce $H(X) = 2.61995$ random bits per pixel. This experiment demonstrates the theory that temperature increases entropy.

A.7 Future Work

Future work could include improving on the bounds of SHA3-512. SHA-3 is based on “wide random permutation” [122]. There are other wide permutations [123] possible. Future work could prove that creating a permutation of the entire image and then choosing a subset of bits whose length is equivalent to the amount of Shannon Entropy in the image resulting in an even more straightforward design and higher security bounds than SHA-3 512.

Further analysis of the pixels is warranted. Justification of how many images are necessary to calibrate the system would be valuable. Measuring the entropy over a more extensive temperature range and researching could show an exponential increase with temperature. Determine if there is a correlation between neighboring pixels.

A.8 Acknowledgement

We thank Bill Soley for his assistance with the temperature control electronics. We thank Stephen Wright of Protolabs for helping with the 3D printed parts.

A.9 Conclusion

We conclude that it is possible to create a low-cost true random number generator. We have demonstrated that the theory that temperature affects entropy happens in commercial CMOS image sensors. We have shown that devices that already have a CMOS image sensor already have a valuable entropy source at their disposal. While we do not assume that we have solved random number generation in the real world, organizations concerned about "getting it right" can use this work as a roadmap. We have shown that exotic, expensive, compute-intensive solutions are not needed to produce high-quality random numbers.

Bibliography

[1] Wesley Chou. Inside SSL: the secure sockets layer protocol. *IT professional*, 4(4):47–52, 2002.

[2] Hannah Davis and Felix Günther. Tighter proofs for the SIGMA and TLS 1.3 key exchange protocols. In *19th International Conference on Applied Cryptography and Network Security (ACNS 2021)*, 2021.

[3] Jon D. Paul. The Scandalous History of the Last Rotor Cipher Machine. In *IEEE Spectrum*. IEEE, AUG 2021.

[4] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A standardized back door. In *The New Codebreakers*, pages 256–281. Springer, 2016.

[5] IETF. Internet Engineering Task Force (IETF). https://www.ietf.org.

[6] John Markoff. Flaw Found in an Online Encryption Method. *New York Times*, JAN 14, 2012.

[7] Arjen K Lenstra, James P Hughes, Maxime Augier, Joppe W Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In *Advances in Cryptology–CRYPTO 2012*, pages 626–642. Springer, 2012.

[8] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, volume 8, 2012.

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

[10] Jonathan Kilgallin and Ross Vasko. Factoring RSA Keys in the IoT Era. In *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 184–189. IEEE, 2019.

[11] Nicolas T Courtois, Daniel Hulme, Kumail Hussain, Jerzy A Gawinecki, and Marek Grajek. On bad randomness and cloning of contactless payment and building smart cards. In *2013 IEEE Security and Privacy Workshops*, pages 105–110. IEEE, 2013.

[12] Adam Young and Moti Yung. Kleptography from standard assumptions and applications. In *International Conference on Security and Cryptography for Networks*, pages 271–290. Springer, 2010.

[13] Moti Yung. Private communication, SEP 20, 2020.

[14] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[15] David Ahmad. Two Years of Broken Crypto: Debian's Dress Rehearsal for a Global PKI Compromise. *IEEE Security & Privacy*, 6(5):70–73, 2008.

[16] Susan Landau. On NSA's Subversion of NIST's Algorithm. https://www.lawfareblog.com/nsas-subversion-nists-algorithm, JUL 25, 2015.

[17] Peter YA Ryan, David Naccache, and Jean-Jacques Quisquater. *The new codebreakers: essays dedicated to David Kahn on the occasion of his 85th birthday*, volume 9100. Springer, 2016.

[18] Susan Landau. Highlights from making sense of Snowden, part II: What's significant in the NSA revelations. *IEEE Security & Privacy*, 12(1):62–64, 2014.

[19] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, and Hovav Shacham. On the Practical Exploitability of Dual EC DRBG in TLS Implementations. 2014.

[20] The many flaws of Dual_EC_DRBG. Blog article, author=Green, M, month=SEP 18,, year=2013.

[21] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz. Reverse-Engineering a Cryptographic RFID Tag. In *USENIX security symposium*, volume 28, 2008.

[22] 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 (TISSEC)*, 13(1):1–32, 2009.

[23] Mike Bendel. Hackers describe PS3 security as epic fail, gain unrestricted access. *Exophase. com*, pages 29–12, 2010.

[24] Bill Buchanan. Not Playing Randomly: The Sony PS3 and Bitcoin Crypto Hacks. https://medium.com/asecuritysite-when-bob-met-alice/not-playing-randomly-the-sony-ps3-and-bitcoin-crypto-hacks-c1fe92bea9bc, NOV 12, 2018.

[25] Bart Preneel. Comments on the NIST Cryptographic Standards and Guidelines Development Program. *NIST Cryptographic Standards and Guidelines Development Process*, JUL 5, 2014.

[26] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012.

[27] Thomas Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). *Internet Engineering Task Force RFC*, 6979:1–79, 2013.

[28] Christopher Ambrose, Joppe W Bos, Björn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. Differential attacks on deterministic signatures. In *Cryptographers' Track at the RSA Conference*, pages 339–353. Springer, 2018.

[29] Bitcoin. Android Security Vulnerability. https://bitcoin.org/en/alert/2013-08-11-android, AUG 11, 2013.

[30] Alex Klyubin. Some SecureRandom Thoughts. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html, AUG 14, 2013.

[31] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of Android openSSL's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 659–668, 2013.

[32] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Practical effect of the predictability of android openSSL PRNG. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 98(8):1806–1813, 2015.

[33] Niels Ferguson. Authentication weaknesses in GCM. *Comments submitted to NIST Modes of Operation Process*, pages 1–19, 2005.

[34] Antoine Joux. Authentication failures in NIST version of GCM. *NIST Comment*, page 3, 2006.

[35] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[36] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[37] George Barwood. Digital signatures using elliptic curves. http://groups.google.com/group/sci.crypt/msg/b28aba37180dd6c6, JAN 7, 1997.

[38] Shay Gueron, Adam Langley, Yehuda Lindell. Webpage for the AES-GCM-SIV Mode of Operation. https://cyber.biu.ac.il/aes-gcm-siv/.

[39] Wikipedia contributors. Server name indication — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Server_Name_Indication&oldid=1051951903, 2021.

[40] C Huitema. Issues and Requirements for Server Name Identification (SNI) Encryption in TLS. Technical report.

[41] CFRG. Crypto Forum Research Group (CFRG). https://irtf.org/cfrg.

[42] IRTF. Internet Research Task Force (IRTF). https://www.irtf.org.

[43] Liliya Akhmetzyanova, Cas Cremers, Luke Garratt, Stanislav Smyshlyaev, and Nick Sullivan. Limiting the impact of unreliable randomness in deployed security protocols. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 277–287. IEEE, 2020.

[44] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In *International conference on provable security*, pages 1–16. Springer, 2007.

[45] Cas Cremers. Private communication, NOV 7, 2021.

[46] Wikipedia contributors. Internet traffic — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Internet_traffic&oldid=1032078480, 2021.

[47] Zeek: An Open Source Network Security Monitoring Tool. https://zeek.org, 2019.

[48] John Althouse. TLS Fingerprinting with JA3 and JA3S. https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967, JAN 15, 2019.

[49] Myron Tribus and Edward C McIrvine. Energy and information. *Scientific American*, 225(3):179–190, 1971.

[50] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. *http://www.stat.fsu.edu/pub/diehard/*, 2008.

[51] Robert G Brown, Dirk Eddelbuettel, and David Bauer. Dieharder. *Duke University Physics Department Durham, NC*, 2018.

[52] Nadia Heninger. Private communication, JAN 20, 2020.

[53] Philippe Flajolet and Andrew M Odlyzko. Random mapping statistics. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 329–354. Springer, 1989.

[54] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[55] Wikipedia contributors. Dot (graph description language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DOT_(graph_description_language)&oldid=1014394658, 2021.

[56] David Garske. Deprecate CyaSSL library #151. https://github.com/cyassl/cyassl/pull/151, OCT 14, 2021.

[57] Chris Conlon, *et al.* CyaSSL. https://github.com/cyassl/cyassl, 2014.

[58] Lawrence You. Private communication, MAY 25, 2021.

[59] IT security scientists at University of Paderborn. Internet-wide research study. https://tls-crawler3.cs.uni-paderborn.de, 2021.

[60] Robert Merget. Private communication, JUN 17, 2021.

[61] Hilton Collins. Is Open Source Software More Secure than Proprietary Products? https://www.govtech.com/security/is-open-source-software-more-secure.html, JUL 26, 2010.

[62] Synopsys Inc. OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html, 2021.

[63] Nicole Forsgren. The 2020 State of the Octoverse. https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf, 2020.

[64] Adam Young and Moti Yung. *Malicious cryptography: Exposing cryptovirology*. John Wiley & Sons, 2004.

[65] Adam Young and Moti Yung. The dark side of "black-box" cryptography or: Should we trust capstone? In *Annual International Cryptology Conference*, pages 89–103. Springer, 1996.

[66] Sumit Singh Dhanda, Brahmjit Singh, and Poonam Jindal. Lightweight cryptography: A solution to secure IoT. *Wireless Personal Communications*, 112(3):1947–1980, 2020.

[67] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[68] Yun Shen and Pierre-Antoine Vervier. IoT security and privacy labels. In *Annual Privacy Forum*, pages 136–147. Springer, 2019.

[69] A. Wool. A note on the fragility of the "Michael" message integrity code. *IEEE Transactions on Wireless Communications*, 3(5):1459–1462, 2004.

[70] Avi Wigderson. On the nature of the Theory of Computation (ToC). In *Electron. Colloquium Comput. Complex.*, volume 25, page 72, 2018.

[71] Merriam-Webster Staff et al. *Merriam-Webster's collegiate dictionary*, volume 2. Merriam-Webster, 2004.

[72] Wikipedia contributors. Decisional Diffie–Hellman assumption — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Decisional_Diffie%E2%80%93Hellman_assumption&oldid=888406260, 2019.

[73] Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):161–185, 2000.

[74] Wikipedia contributors. Key encapsulation — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Key_encapsulation&oldid=1029672752, 2021.

[75] Eric M Hutchins, Michael J Cloppert, Rohan M Amin, et al. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1(1):80, 2011.

[76] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-28. https://www.rfc-editor.org/rfc/pdfrfc/rfc8446.txt.pdf, 2018. Internet Engineering Task Force (IETF).

[77] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[78] Virgil D Gligor. *A guide to understanding covert channel analysis of trusted systems*, volume 30. National Computer Security Center, 1994.

[79] Anthony Rutkowski. Creating TLS: The Pioneering Role of Ruth Nelson. https://circleid.com/posts/20190124_creating_tls_the_pioneering_role_of_ruth_nelson/, JAN 24, 2019.

[80] Whitfield Diffie, Paul C Van Oorschot, and Michael J Wiener. Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, 2(2):107–125, 1992.

[81] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[82] Wikipedia contributors. Key-agreement protocol — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Key-agreement_protocol&oldid=1033357099, 2021.

[83] Google Cloud. SSL Proxy Load Balancing overview. https://cloud.google.com/load-balancing/docs/ssl.

[84] Don Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM journal of research and development*, 38(3):243–250, 1994.

[85] Sharon Goldberg. Surveillance without borders: The "traffic shaping" loophole and why it matters. *The Century Foundation*, 2017.

[86] Raspberry Pi. Raspberry Pi High Quality HQ Camera - 12MP. https://cdn-shop.adafruit.com/970x728/4561-12.jpg.

[87] John Von Neumann. Various techniques used in connection with random digits. *Appl. Math Ser*, 12(36-38):5, 1951.

[88] Claude Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[89] Elaine B Barker, John Michael Kelsey, et al. *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, National Institute of Technology, 2007.

[90] Sony. IMX477. https://www.sony-semicon.co.jp/products/common/pdf/IMX477-AACK_Flyer.pdf.

[91] Laird Thermal Systems, Inc. HiTemp ETX Series ETX3-3-F2-1518-TA-W6. https://www.lairdthermal.com/datasheets/datasheet-ETX3-3-F2-1518-TA-W6.pdf.

[92] Stuart Bennett. A brief history of automatic control. *IEEE Control Systems Magazine*, 16(3):17–25, 1996.

[93] Michael Luby. Pseudorandomness and Cryptographic Applications. *JSTOR*.

[94] Christof Paar and Jan Pelzl. *Understanding Cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[95] Benjamin Jun and Paul Kocher. The Intel random number generator. *Cryptography Research Inc. white paper*, 1999.

[96] Mike Hamburg, Paul Kocher, and Mark E Marson. Analysis of Intel's Ivy Bridge digital random number generator. *Online: http://www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf*, 2012.

[97] Daniel J Bernstein et al. ChaCha, a variant of Salsa20. In *Workshop record of The State of the Art Stream Ciphers*, volume 8, pages 3–5, 2008.

[98] Theodore Ts'o. Random: replace non-blocking pool with a Chacha20-based CRNG. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e192be9d9a30555aae2ca1dc3aad37cba484cd4a, 2016.

[99] Qiushi Wu and Kangjie Lu. On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commits. *To be published*, 2021.

[100] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, Cham, 2020. Springer International Publishing.

[101] Gaëtan Leurent and Thomas Peyrin. SHA–1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. Cryptology ePrint Archive, Report 2020/014, 2020. https://eprint.iacr.org/2020/014.

[102] Bruce Schneier. Complexity the Worst Enemy of Security. *Interview with Computer World Hong Kong (CWHK)*, 17, 2012.

[103] Werner Schindler. Random number generators for cryptographic applications. In *Cryptographic Engineering*, pages 5–23. Springer, 2009.

[104] Dice–O–Matic hopper and elevator - GamesByEmail. http://gamesbyemail.com/news/diceomatic.

[105] Jim Cheetham. OneRNG, Open Hardware Random Number Generator. http://onerng.info/.

[106] D. Schellekens, B. Preneel, and I. Verbauwhede. FPGA Vendor Agnostic True Random Number Generator. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, AUG 2006.

[107] Thomas Jennewein, Ulrich Achleitner, Gregor Weihs, Harald Weinfurter, and Anton Zeilinger. A fast and compact quantum random number generator. *Review of Scientific Instruments*, 71(4):1675–1680, 2000.

[108] John Walker. HotBits: Genuine random numbers, generated by radioactive decay. https://www.fourmilab.ch/hotbits/, 1996.

[109] Bruno Sanguinetti, Anthony Martin, Hugo Zbinden, and Nicolas Gisin. Quantum Random Number Generation on a Mobile Phone. *Phys. Rev. X*, 4:031056, SEP 2014.

[110] Kenji Irie, Ian M Woodhead, Alan E McKinnon, and Keith Unsworth. Measured effects of temperature on illumination-independent camera noise. In *2009 24th International Conference Image and Vision Computing New Zealand*, pages 249–253. IEEE, 2009.

[111] R. D. Gow, D. Renshaw, K. Findlater, L. Grant, S. J. McLeod, J. Hart, and R. L. Nicol. A Comprehensive Tool for Modeling CMOS Image-Sensor-Noise Performance. *IEEE Transactions on Electron Devices*, 54(6):1321–1329, JUN 2007.

[112] Eric R Fossum et al. CMOS image sensors: electronic camera-on-a-chip. *IEEE Transactions on Electron Devices*, 44(10):1689–1698, 1997.

[113] EMVA Standard 1288: Standard for Characterization of Image Sensors and Cameras. http://www.emva.org/wp-content/uploads/EMVA1288-3.0.pdf.

[114] Helmuth Spieler. *Semiconductor Detector Systems.* Oxford University Press, 1 edition, 2005.

[115] FIPS PUB 140-2, Security Requirements for Cryptographic Modules, 2002. U.S.Department of Commerce/National Institute of Standards and Technology.

[116] Steve H. Weingart. *Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses*, pages 302–317. Springer Berlin Heidelberg, Berlin, Heidelberg, AUG 17–18, 2000.

[117] Mario Stipčević and Çetin Kaya Koç. True random number generators. In *Open Problems in Mathematics and Computational Science*, pages 275–315. Springer, 2014.

[118] Jack from AlmaPhoto. OPENING RASPBERRY PI HIGH QUALITY CAMERA RAW FILES. https://www.strollswithmydog.com/open-raspberry-pi-high-quality-camera-raw/.

[119] Boaz Barak, Ronen Shaltiel, and Eran Tromer. True random number generators secure in a changing environment. In *Cryptographic hardware and embedded systems-CHES 2003*, pages 166–180. Springer, 2003.

[120] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[121] Andrew DeOrio, Daya Shanker Khudia, and Valeria Bertacco. Post-silicon bug diagnosis with inconsistent executions. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 755–761, 2011.

[122] Wikipedia contributors. SHA-3 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=SHA-3&oldid=1026512393, 2021.

[123] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.