# Pseudorandom Black Swans: Cache Attacks on CTR_DRBG

Shaanan Cohney[1], Andrew Kwong[2], Shahar Paz[3], Daniel Genkin[2], Nadia Heninger[4], Eyal Ronen[5], Yuval Yarom[6]
[1]*University of Pennsylvania*, shaananc@seas.upenn.edu
[2]*University of Michigan*, {ankwong,genkin}@umich.edu
[3]*Tel Aviv University*, shaharps@tau.ac.il
[4]*University of California, San Diego*, nadiah@cs.ucsd.edu
[5]*Tel Aviv University and COSIC (KU Leuven)*, er@eyalro.net
[6]*University of Adelaide and Data61*, yval@cs.adelaide.edu.au

***Abstract*—Modern cryptography requires the ability to securely generate pseudorandom numbers. However, despite decades of work on side-channel attacks, there is little discussion of their application to pseudorandom number generators (PRGs). In this work we set out to address this gap, empirically evaluating the side channel resistance of common PRG implementations.**

**We find that hard-learned lessons about side channel leakage from encryption primitives have not been applied to PRGs, at all levels of abstraction. At the design level, the NIST-recommended CTR_DRBG design does not have forward security if an attacker is able to compromise the state via a side-channel attack. At the primitive level, popular implementations of CTR_DRBG such as OpenSSL's FIPS module and NetBSD's kernel use leaky T-table AES as their underlying block cipher, enabling cache side-channel attacks. Finally, we find that many implementations make parameter choices that enable an attacker to fully exploit the side-channel attack in a realistic scenario and recover secret keys from TLS connections.**

**We empirically demonstrate our attack in two scenarios. In the first, we carry out an asynchronous cache attack that recovers the private state from vulnerable CTR_DRBG implementations under realistic conditions to recover long-term authentication keys when the attacker is a party in the TLS connection. In the second scenario, we show that an attacker can exploit the high temporal resolution provided by Intel SGX to carry out a *blind* attack to recover CTR_DRBG's state within three AES encryptions, without viewing output, and thus to decrypt passively collected TLS connections from the victim.**

## I. Introduction

It is a truth universally acknowledged, that a securely implemented cryptographic primitive must be in want of a cryptographically secure pseudorandom number generator [3]. Modern cryptography relies on randomness to prevent an attacker from predicting secret values generated by parties in a cryptographic protocol. Indeed, random values are universally used to ensure security properties for nearly all cryptographic data, including secret keys for confidentiality or integrity, secret keys for public-key encryption, key exchange, or signatures, as well as for protocol nonces to prevent replay attacks.

Thus, a cryptographically secure Pseudorandom Generator (PRG) is one of the fundamental primitives of modern cryptography, both in theory and in practice.

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

A number of academic works and practical security failures have illustrated the disastrous effects on real-world cryptography from flawed random number generation implementations or designs in the wild. These have ranged from unintentional flaws such as failure to properly seed PRGs [34, 45, 52, 94], to designs prone to implementation mistakes [19], to a suspected intentional back door in the now "deprecated and disgraced" [62] Dual EC DRBG design, which appears to have been repurposed and exploited in the wild [17, 18].

Since their introduction in the seminal works of [5, 64, 65], microarchitectural attacks that exploit contention on internal components to leak information have been used to violate nearly every security guarantee offered by computer systems. Indeed, in recent years there have numerous examples of side-channel attacks with diverse targets and vectors. These range from attacks that extract cryptographic keys from keystroke timing [28, 92] via CPU caches, attacks that exploit transient execution for breaking fundamental OS isolation guarantees [16, 46, 51, 81, 85], and even attacks that exploit limitations in memory hardware to change or read the contents of stored data [14, 42, 44, 47, 48]. Side-channel resistance is among the key security properties demanded of implementations.

Much less is known, however, about the security of PRGs in the presence of side-channel leakage. While backtracking resistance and prediction resistance are stated to be among the main security goals of the designs in NIST's PRG recommen-

dations (NIST SP 800-90A), the standard does not consider the impact of side channel attacks on these goals. Although some initial evidence [97] already indicates the possibility of exploiting side-channel vulnerabilities in PRG seeding, a systematic exploration of side channel leakage from PRG implementations has not been performed. Thus, in this paper we set out to explore the following main question:

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

### A. Our Contribution

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

**The Use of T-Table AES.** T-table AES is a performance-oriented AES implementation that uses table lookups to compute the state transitions between individual encryption rounds. Unfortunately, because these lookups are key-dependent, T-table AES has become the canonical example of cache side channel leakage [10, 58, 64].

While the use of T-table AES for encryption and decryption operations has been greatly reduced in light of the threat posed by side channels and the availability of AES-NI hardware, similar lessons do not seem to have been learned for the case of random number generation. Remarkably, even after more than a decade of attacks, [5, 13, 31, 57, 64] we show that unprotected and leaky T-tables are still used for encrypting the counter inside CTR_DRBG by the following popular implementations:

- The OpenSSL 1.0.2 FIPS Module uses T-Table AES for CTR_DRBG. We note that use of this library is the *only* way to obtain U.S government certification for a cryptographic module without submitting to the expensive and time-consuming validation process.
- The NetBSD kernel uses CTR_DRBG with a T-Table AES implementation as the system-wide random number generator.
- The FortiOSv5 network device operating system uses the same vulnerable CTR_DRBG implementation as NetBSD.
- mbedTLS-SGX, a port of the popular mbedTLS cryptography library to SGX [95].
- The nist_rng library [39], which is a library for random number generation used by open source projects such as libuntu (a C implementation of NTRUEncrypt), the XMHF hypervisor, as well as others.

**CTR_DRBG State Recovery.** By adapting previous work on AES encryption [58] to the PRG setting, we extend the work of Woodage and Shumow [86] to show how an attacker who observes the cache access patterns of CTR_DRBG-based random number generation can recover the PRG's state using about 2000 bytes of the PRG's output. We then empirically demonstrate how a client that connects to a malicious TLS sever can be coerced to provide enough PRG output that an attacker can recover the PRG state used during the TLS handshake by concurrently observing the PRG's cache access patterns.

**Extracting the Client's TLS Authentication Keys.** Next, we show that NetBSD's kernel, OpenSSL's FIPS module and FortiOS fail to reseed the PRG with a sufficient amount of entropy. Thus, by using a moderate amount of brute forcing for the client entropy, the attacker can wind forward the client's PRG and recover the ECDSA nonce used by the client to authenticate herself to the malicious TLS server. Finally, using the recovered ECDSA nonce and the signature produced by the client during the TLS handshake, the attacker can recover the client's long term authentication keys. With authentication key in hand, the attacker can impersonate the client in future TLS connections.

**State Recovery Without a Malicious TLS Server.** The above attack on TLS requires the victim client to connect to a malicious TLS server, allowing the attacker to observe sufficient output generated by the client's CTR_DRBG implementation while simultaneously observing the client's cache access patterns across many AES encryption operations. Tackling this limitation, we perform a a novel differential cryptanalysis attack exploiting side channel leakage from T-table based CTR_DRBG running inside an SGX enclave. This attack leverages the fact that CTR_DRBG encrypts an incrementing counter. Our technique is capable of extracting the PRG's state from only three AES encryption operations, without requiring the attacker to observe the PRG's output. Thus, we eliminate the need for the TLS client to connect to an attacker-controlled server. We also note that this type of attack might also be applicable to other settings with similar constraints such as GCM-SIV [30].

**Breaking TLS Connections With High-Entropy PRG Reseeding.** Finally, we note that any call to CTR_DRBG for random byte generation must use at least three AES encryption operations, and thereby produce the cache access information required by our differential cryptanalysis state-recovery technique. Since we no longer require the TLS client to connect to an attacker-controlled server, this results in

an attack that recovers the PRG state on *any* request for random bytes, regardless of how the implementation reseeds the PRG. We demonstrate recovery of the premaster secret, master secret, and symmetric encryption keys for any TLS connection made by mbedTLS-SGX (a port of mbedTLS to SGX [95]) to any TLS server. In particular, we are able to passively decrypt the session by observing cache access patterns made by mbedTLS-SGX.

**Summary of Contributions.** In this work we study the implications of side channel analysis on random number generation. Our contributions can be summarized as follows.

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

### B. Coordinated Disclosure

We disclosed the vulnerabilities we discovered to the security teams of OpenSSL, Fortinet, and NetBSD in May 2019. OpenSSL responded that these attacks are outside their threat model. Both NetBSD and Fortinet have since shared advisories and remediations for their customers. The DRBG flaw in FortiOS was assigned CVE-2019-15703.

## II. Background

### A. Pseduorandom Generators

The term "DRBG" does not seem to be widely used outside of the government context, so for the purposes of this paper, we will use the term pseudorandom generator (PRG). We begin by providing basic background regarding pseudorandom generators and their security properties. Informally, a PRG is an algorithm that, given an initial seed, produces a stream of random bits such that an attacker cannot distinguish the produced stream from a truly uniform random bit stream with probability better than some negligible bound.

**PRG Definition.** Following Dodis *et al.* [21] and Woodage and Shumow [86], a *PRG with input* is a triplet of polynomial time deterministic algorithms $\{\mathsf{instantiate}, \mathsf{generate}, \mathsf{reseed}\}$. The PRG is instantiated by calling instantiate on an entropy sample $I$ and a nonce $N$, and outputs initial state $S_0$. Next, generate gets as input a state $S$, a number of bits to output $nbits$, an additional input $addin$, and outputs new state $S'$ and bits $R \in \{0,1\}^{nbits}$. Finally, reseed gets as input a state $S$, an entropy sample $I$, an additional input $addin$, and outputs a new state $S'$.

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

**PRG Security.** Woodage and Shumow [86] define three security properties for a PRG: robustness, backtracking resistance, and prediction resistance. Backtracking resistance is the property that if the generator is compromised at time $t_1$, an adversary remains unable to distinguish outputs generated prior to $t_1$ from random. Similarly, prediction resistance ensures that there is some time $t_2$ after $t_1$ when no further outputs can be distinguished from random. Robustness incorporates both of these guarantees into a single property.

Next, while the model of Dodis *et al.* [21] and Woodage and Shumow [86] includes an attacker that is able to compromise the entropy distribution used for sampling entropy to the PRG, we consider a weaker attacker who is unable to do so.[1]

We instead assume that the PRG correctly receives entropy samples drawn uniformly at random from the entropy space, better matching our real-world scenario.

Finally, as our attack targets the prediction resistance guarantee of CTR_DRBG, we now provide a more formal definition for prediction resistance, from Dodis *et al.* [21].

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

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

An adversary's advantage, and therefore the security strength of the PRG, is parameterized by the number of calls an

[1] We therefore obtain a stronger result as our weaker attacker is able break the PRG despite her inability to corrupt the entropy source.

adversary makes to the above procedures along with the adversary's probability of successfully guessing the challenge bit in the **next-ror** game. We use the following formal security definition for a PRG:

**Definition 1** (PRG with Input Security)**.** *A PRG with input $\mathcal{G}$ is called a $(t, q_D, q_R), \delta-$prediction-resistant PRG if for any adversary $\mathcal{A}$ running in time at most t, making at most $q_D$ calls to* update *with $q_R$ calls to* next-ror/get-output*, and one call to* get-state*, which is the last call $\mathcal{A}$ is allowed to make prior to calling* next-ror*, it holds that*

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

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

### B. NIST SP 800-90 and Related Standards

NIST Special Publication (SP) 800-90 is entitled "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" and is the de facto standard for algorithms for generating random numbers. The document was first published in 2006 and has undergone three revisions: "800-90 Revised", published in 2007, "800-90 A", published in 2012, and "800-90A Rev. 1", published in 2015. The first three publications contained four pseudorandom number generator designs, while the last publication contained only three. The missing design was the infamous DualEC DRBG, which was removed from the publication after Shumow and Ferguson discovered a design flaw that enabled a backdoor [77] which was later confirmed by Snowden [66]. The three remaining designs in NIST 800-90A Rev. 1 are HMAC_DRBG, HASH_DRBG and CTR_DRBG, which are based on HMAC, hash, and block cipher primitives respectively. For the remainder of this paper, we will refer to the 2015 publication as SP 800-90A.

### C. AES

AES encryptions and decryptions can be decomposed into four operations (AddRoundKey, SubBytes, ShiftRows, and MixColumns). Performance-optimized software implementations usually use a series of lookup tables known as "T-tables" to combine the latter three operations. AES encryptions and decryptions can be decomposed into rounds, which use round keys derived from the secret key to transform the input into a sequence of states. The state at each round is used to index into the T-tables, and the results are XORed with the round key to produce the state for the next round. The final round of AES uses a different T-table from earlier rounds as there is no MixColumns operation in that round. Unfortunately, by observing the memory access patterns to these tables, an attacker can recover the cipher's secret key within only a few encryptions. Indeed, starting from [64], there has been a large body of work on attacking table-based AES implementations [28, 31, 37, 78, 96].

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

### D. Cache Attacks

Our work contributes to a long line of cache-based side-channel attacks. These attacks have yielded varied and robust mechanisms [20, 29, 79] for breaking cryptographic schemes using information leakage from cache timings. Popular targets have included digital signature schemes [4, 27] and symmetric ciphers [64, 68, 92], despite the inclusion of countermeasures in popular cryptographic implementation libraries [23, 70]. Recent literature has also begun to examine side-channel vulnerabilities in environments provided by trusted processor enclaves, particularly Intel SGX [11, 49, 55, 81, 88, 89], which are designed to be more secure against even local attackers who are able to run unprivileged code.

**Flush+Reload.** Flush+Reload is a side-channel attack technique that consists of three steps. In the first step, the attacker *flushes* or evicts a memory location from the cache. The attacker then waits a while, allowing the victim to execute. Finally, in the third step, the attacker *reloads* the monitored memory location and measures the reload time. If the victim has accessed the memory location between the flush and the reload steps, the location will be cached, and the reload will be fast. Otherwise, the memory will not be cached and the reload will be slow. Flush+Reload has been used to attack symmetric [37] and public key [4, 22, 27, 67, 92] cryptography, as well as for non-cryptographic and speculative execution attacks [28, 46, 51, 54, 81, 82, 85, 90] attacks.

**Prime+Probe.** While powerful, Flush+Reload relies on the victim and the attacker accessing the same memory location and is thus typically applied to OS-deduplicated pages in binaries and shared libraries. When shared memory is not available (e.g., for SGX), we use a different cache attack technique called Prime+Probe [64, 79].

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

## III. CTR_DRBG

CTR_DRBG is a PRG design described in NIST SP 800-90A. It uses the encryption of an incrementing counter under a block cipher to generate outputs. The block cipher may be either 3DES with a 64-bit key or AES with a key of length 128, 192, or 256 bits. The design mixes in additional data at various stages. A derivation function (commonly the same

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

```
1:  function UPDATE(K, V, addin)
2:      temp ← null
3:      while len(temp) < seedlen do
4:          V ← (V + 1) mod 2^blocklen
5:          output_block ← encrypt(K, V)
6:          temp ← temp||output_block
7:      temp ← temp ⊕ addin
8:      K′ ← leftmost(temp, keylen)
9:      V′ ← rightmost(temp, blocklen)
10:     return K′, V′
```

block cipher under a different key) can optionally be used to extract entropy from the additional data. The implementations we examined all used a derivation function.

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

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

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

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

**State Update.** Each of CTR_DRBG's functions call a subroutine update, outlined in Algorithm 1, that updates the internal state. The routine's input is a key $K$, counter $V$, and additional data $addin$. In Lines 4–6 the function increments the counter $V$ and appends the encryption of $V$ under key $K$ to a buffer $temp$. This process is repeated until $temp$ contains $seedlen$ bytes. The resulting buffer is then XORed with $addin$ (Line 7). Finally, in Lines 8–9 the function outputs the new key $K'$ as the leftmost $keylen$ bits of the buffer, and new counter value $V'$ as the rightmost $blocklen$ bits of the buffer, where $blocklen$ is the block length of the cipher.

**Generating a Random Stream.** A user generates output from the PRG by calling the generate function outlined in Algorithm 2. It takes as input the state $S$, the number of bits requested $nbits$, and a string $addin$, and outputs a string $nbits$ in length and an updated state $S'$. According to SP 800-90A, the $addin$ parameter "may be a means of providing more entropy for the DRBG internal state". This additional input is allowed to be public or private and may contain secrets if private. The specification notes that "if the additional input is kept secret and has sufficient entropy, the input can provide more assurance when recovering from the compromise of the entropy input, the seed or one or more DRBG internal states". However, the specification does not include requirements for either secrecy or entropy for $addin$.

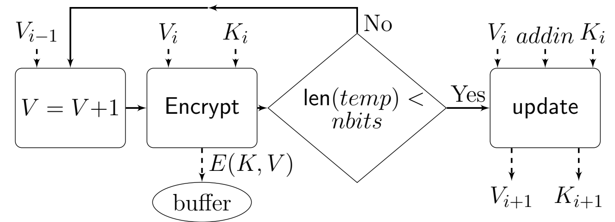


Fig. 1: The central loop of the generate function increments the counter $V$, encrypts $V$ under $K$, and adds the output to a buffer $temp$, repeating until $nbits$ have been generated. The function then updates the key and state before returning the contents of the buffer.

The generate function first checks if a reseed is needed, and if so, throws an error[2] (Lines 3–4).

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

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

**Reseeding.** The reseed function is intended to ensure that high quality entropy is mixed into the state as required.

[2]While the inclusion of an error message does not strictly adhere to our PRG definition, following Woodage and Shumow [86] we assume inputs are valid and omit consideration of errors from our analysis.

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

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

The reseed function takes as input additional input $addin$, an entropy sample $I$, and a state $S$ that consists of the key $K$, counter $V$, and reseed counter $c$. It calls the update subroutine on a derivation function taken over $I$ and $addin$, which updates $K$ and $V$. Finally, it resets the reseed counter $c$ to 1 and returns the new key, counter, and reseed counter.

### A. Cryptanalysis of CTR_DRBG

**DRBG Security Proofs.** Woodage and Shumow [86] note that historical analyses of the security claims in SP 800-90A [15, 35, 41, 75, 76, 93] were limited by simplifying assumptions that were believed to be necessary due to nonstandard elements of the designs. Their analysis evaluated the standard's claims that the designs in the standard are both "backtracking resistant" and "prediction resistant". They provide robustness proofs that include backtracking and prediction resistance for both the HMAC and hash constructions, but were unable to do so for CTR_DRBG and instead identified an attack against the prediction resistance property.

**Attacking CTR_DRBG.** Bernstein [6] notes that to obtain prediction resistance after every random bit, the generate process must be called with only a single bit, incurring massive performance costs. Furthermore, SP 800-90A notes that "For large generate requests, CTR_DRBG produces outputs at the same speed as the underlying block cipher algorithm encrypts data". Woodage and Shumow [86] use this observation to propose an attack scenario where large amounts of CTR_DRBG output is buffered, setting the stage for a side channel attack on the block cipher key. They give the following procedure for recovering output at $t+1$ from output $r_t$ and key $K_t$ that was compromised at time $t$:

1) **Counter Recovery From Output.** Attacker computes the state prior to the last update as $V'_t = \mathsf{decrypt}(K_t, r_t)$
2) **Generating $S_{t+1}$.** The attacker winds the generator forward by computing $K_{t+1}, V_{t+1} = \mathsf{update}(K_t, V'_t, addin_t)$
3) **Generating PRG Output $r_{t+1}$.** This state is now used to compute $r_{t+1} = \mathsf{generate}(K_{t+1}, V_{t+1}, addin_{t+1})$

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

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

## IV. State Recovery Attack

We show that the attack described by Woodage and Shumow [86] is practical by recovering the CTR_DRBG state variables $K$ and $V$ via a cache side-channel attack against the underlying AES implementation. We begin with an overview of the popular implementations we analyze in this section.

### A. Implementation Deep Dives

We examined the CTR_DRBG parameter choices of four implementations representing diverse use cases: the NetBSD operating system, the Fortinet FortiVM virtualized network device, and two versions of the OpenSSL cryptographic library.

**FortiOS.**

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

After reverse-engineering the operating system binaries, we discovered that FortiOSv5 replaces Linux's default implementation of `/dev/urandom` with the nist_rng library [39]. We note that Cohney *et al.* [19] analyzed FortiOSv4 and found that it behaved similarly, replacing the system's default PRG with a FIPS certified design. Both FortiOS v4 and v5 use OpenSSL to provide basic cryptographic functionality, which in turn relies on `/dev/urandom`. While the original OpenSSL will use an AES hardware implementation if it is available, Fortinet's override makes OpenSSL fall back to an unprotected T-table-based AES implementation based on the nist_rng library.

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

**NetBSD.** The NetBSD operating system uses CTR_DRBG as the default source of system randomness. The kernel uses the nist_rng library with 128-bit AES as the default underlying cipher. We examined the kernel source code and single-stepped through a running kernel to verify our findings. As in the FortiOS case, the AES implementation is software-based with unprotected T-Table accesses, based on the nist_rng library.

On each generate call, the state is updated using additional entropy from `rdtsc`, a high resolution CPU counter. NetBSD schedules a reseed after $2^{30}$ calls to the PRG. Notably, the reseed counter is incremented after each request to the PRG, rather than after generation of each block. This provides an opportunity for the attacker to gather a large quantity of PRG output and leakage traces with the same key, before CTR_DRBG is reseeded.

**OpenSSL FIPS Module.** We examined the OpenSSL FIPS module, which supports only OpenSSL 1.0.2. This implementation is one of a small number of libraries that a manufacturer can use to be FIPS compliant without submitting the entire product for certification [24]. The module uses CTR_DRBG with a user configurable key length. Notably, while OpenSSL 1.0.2 FIPS uses hardware instructions for AES encryption, the CTR_DRBG implementation uses a lower-level interface for AES. Instead of selecting the best implementation available (as the AES interface used for encryption does), the lower-level interface used by CTR_DRBG uses a hand-coded T-Table AES implementation. On each generate call, the state is updated using the time in microseconds, a counter, and the PID. The FIPS module reseeds the PRG after $2^{24}$ calls to generate.

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

### *B. Side Channel Attacks on AES-128*

T-Table AES is the canonical target for cache side channel attacks. Starting from Bernstein [5] many works [28, 31, 37, 64, 96] have demonstrated key extraction from cache access patterns of table-based implementations.

Since CTR_DRBG uses T-Table AES as its underlying cryptographic primitive, we implemented the attack of Neve and Seifert [58] on the last encryption round of AES in order to extract the AES key from the CTR_DRBG's cache access pattern.

## V. Cache Attack Details

In this section, we present the details of our state recovery attack. In the synchronous model of Osvik *et al.* [64], an attacker observes the plaintext and is able to probe the cache state immediately before triggering an encryption with an unknown key. The attacker is also able to probe the cache state immediately after each encryption. Observing the cache access patterns caused by the first round of AES during a few encryption operations is sufficient to recover the key [64].

**Attacking the Last Round of AES.** Working in the synchronous model of [10, 58, 64] we target the final round of AES, with attacker-observed ciphertext, rather than plaintext.

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

**Handling Missing Information.** While the attack outlined above works when the attacker has perfect visibility over $q_i$ and $i$, on a real system the attacker does not directly observe $q_i$. Instead, she identifies a contiguous set of bytes that are fetched into the cache together (a cache line, typically 64 bytes) and thus loses information about some of the least significant bits of $q_i$. On our test machine, each access corresponded to sixteen different possible values for $q_i$, as each final T-Table byte is stored four times, in a 4-byte integer, sixteen of which are in each cache line. Further, the attacker does not know $i$, as she does not know which cache access produced which ciphertext byte. Thus, in order to obtain a candidate key byte $k_i$, the attacker must somehow *guess* the value of $q_i$ from the table indexes accessed in the last round as well as guess the missing 4 bits from $q_i$. As we expect about 11 distinct indexes to be accessed in the last round [58], this results in about $11 \cdot 2^4 = 176$ candidate values for each $k_i$, out of 256 possible candidates.

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

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

Following [58], the attacker counts cache hits that could correspond to each possible key byte value $k$ from `0x00` to `0xFF` for each position $i$ and stores the count in a table $\mathcal{S}$:

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

with $\ell$ the number of cache lines, $m$ the number of bytes per cache line, and $n$ the number of traces. As analyzed by [10,

58], the $i$-th byte of the last round key is then the value of $k$ such that $\mathcal{S}[i][k]$ is maximal.

### A. Obtaining Trace Data

We describe how we mount Flush+Reload against CTR_DRBG. We begin by recalling that the attack of [58] outlined in Section IV-B requires the attacker to gather ciphertexts paired with the corresponding traces of the cache state following the encryption operation that produced that ciphertext.

**Matching PRG Output.** To recover the AES key, an attacker must match each ciphertext to a trace taken in the interval following the encryption that produced it, but before the subsequent encryption. In the synchronous model of Osvik *et al.* [64] where the attacker triggers encryption operations directly, this matching is trivial. However, in our setting, a request for random bytes initiates a rapid series of encryptions. If the attacker's probes take a long time compared to an encryption operation, the attacker cannot easily interleave probes. This difficulty is exacerbated by the fact that encryptions vary in duration due to other system activity, making the naive strategy of probing at evenly spaced intervals fail to produce matching traces and ciphertext pairs.

**Tickers.** In order to use the synchronous setting analysis of Osvik *et al.* [64], we align traces and ciphertexts by using what we term "tickers". Tickers are frequent cache probes that measure how long it takes to access cache lines that contain program instructions. A cache hit on a ticker gives the attacker a signal she can use to determine whether to probe the cache lines containing the T-Table used in the last AES encryption round. In our case, we set two tickers. The first ticker queries instructions at the start of the encryption code (as loaded into the process' address space), and the second queries instructions at the end of the encryption code. When either ticker is triggered, we probe the T-Table cache lines, ideally measuring cache state before and after encryptions.

**Handling Drift.** While tickers provide some signal, as depicted in Figure 2, variations in how the probe process is scheduled with respect to the victim process introduce imperfections in the signal provided by the tickers. Therefore, we also use timing heuristics to match traces to corresponding ciphertexts. More specifically, we iterate through the traces we collect, and keep a counter identifying the next ciphertext to be matched to a trace. Then, for each trace, we either match it to the current ciphertext and increment the counter or discard it. We base this decision on the accompanying ticker and timing data.

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

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

**Overcoming Prefetching.** Modern CPUs attempt to learn a program's cache access pattern and fetch data into caches before this data is actually needed. This data prefetching frustrates cache side channel attacks against T-Table AES by reducing the extent to which a recorded cache hit corresponds to an actual–rather than predicted–access. If an entire AES T-Table is preemptively fetched into memory, a naïve cache side channel attack will not succeed because the attacker will record cache hits for every memory line.

We mitigated the effect of the prefetcher by accessing cache lines in an irregular order, using the pointer chasing technique of Osvik *et al.* [64]. This reduces the ability of the prefetcher to predict our cache accesses and therefore prefetch those lines.

**Performance Degradation.** If the time it takes to probe the cache state is too long relative to the duration of an encryption, an attacker will not be able to generate traces that accurately capture the state of the cache after each encryption. Allan *et al.* [1] showed that this difficulty could be mitigated by continuously flushing cache lines containing victim program instructions, so that the victim process was significantly slowed down. Flushing cache lines requires the victim to repeatedly fetch code from main memory, increasing access times. On our system, this slowed down the average duration of an encryption from 2 $\mu$sec to 32 $\mu$sec, giving us a large 34$\mu$sec window between successive last AES rounds for cache probing.

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

### B. Evaluation of State Recovery

**Attack Scenario.** Our attack scenario is as follows. First, we assume an attacker who can execute unprivileged code on a target machine. Next, a victim process on the same machine uses CTR_DRBG and makes a call to generate, requesting about 2 KB of pseudorandom output. The attacker then uses Flush+Reload to monitor cache accesses during the AES operations inside the CTR_DRBG, and recovers the PRG state using the techniques described above. Our experimental setup instantiates this scenario in a concrete setting.

**Targeted Software.** We targeted OpenSSL 1.0.2 configured to use the nist_rng library with AES128 as the underlying block

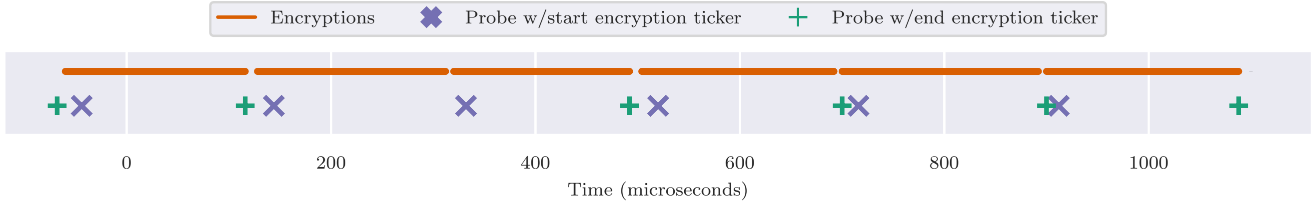


Fig. 2: Probes do not perfectly align with the start and end of encryptions. Ideally, the start and end of an encryption probe should follow shortly after the start and end of an encryption. However, fluctuations in encryption duration and ticker timing accuracy cause misalignments. The problem is illustrated at $\approx 380\mu$s, where no end encryption ticker is visible, and at $\approx 900\mu$s where the end encryption ticker appears past the start of the next iteration.

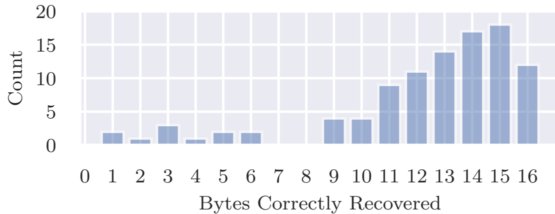


Fig. 3: With the prefetcher enabled, our state recovery technique often recovers only a subset of the full 16-byte AES key. We here depict the frequency with which a given number of bytes were recovered, across 100 trials.

cipher for the PRG. Beyond the implementations mentioned in Section IV-A, the nist_rng library is used by libuntu (a C implementation of NTRUEncrypt), the XMHF hypervisor, among others. As mentioned before, the nist_rng library uses a leaky T-table based AES implementation and does not support `AES-NI` hardware instructions.

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

**Empirical Results.** In 100 trials with the prefetcher disabled we were always able to recover the state, with an average false positive rate of 4.58% and false negative rate of 5.01%. As in Figure 3, with the prefetcher enabled our attack succeeded in 12.0% of trials with average false positive rate 28.5% and false negative rate 1.94%. State recovery took an average of 19s in both cases, with hardware as above.

## VI. Attacking TLS

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

| 00 | 02 | Padding | 00 | 48-byte PMS |
|---|---|---|---|---|

Fig. 4: PKCS#1v1.5 RSA encryption padding appends a pseudorandom padding string to the message, together with some fixed bytes. The padding block is filled with $k-3-\ell$ non-zero bytes that are generated by a pseudorandom number generator, where $k$ is the byte-length of the modulus and $\ell$ is the byte-length of the message to be encrypted.

### A. RSA Background

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

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

**RSA Padding.** An RSA-encrypted key exchange message begins by padding the message using PKCS#1 v1.5 [40] padding as depicted in Figure 4. PKCS#1 v1.5 padding is not CCA-secure and has led to numerous cryptographic attacks against RSA in practice [9, 26]. Yet, it remains by far the most common padding method where RSA encryption is still used, including versions of TLS prior to 1.3.

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

**RSA-PSS.** RSA-PSS is a probabilistic signature scheme with a formal security proof [53]. The padding scheme is designed to avoid the flaws in PKCS#1 v1.5 padding. The scheme uses a sequence of hashing operations and mask generation functions to generate a padded message from a salt $s$ and the input message $m$. The salt can in general be a maximum of $\mathsf{len}(m) + hLen$ bytes in length, where $hLen$ is the length of the hash function output. RFC8446 (August 2018) [72] updates TLS 1.2, adding optional support for RSA-PSS signatures [56], but

specifies that “the length of the Salt MUST be equal to the length of the [digest] output”.

### B. ECDSA

ECDSA is a standardized public key signature algorithm [43]. The global parameters for an ECDSA key pair include a pre-specified elliptic curve $C$ with base point $G$ of order $n$. The signer’s private key is a random integer $1 < d_A < n$ and the public key is $Q = d_A G$.

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

### C. TLS Handshake Protocol

We describe the TLS 1.0, 1.1, and 1.2 handshake protocols necessary for our attack. A TLS handshake begins with a ClientHello message containing a 32-byte nonce along with a list of supported cipher suites. The standard specifies that the nonce should consist of a four-byte timestamp and 28 bytes of raw output from a pseudorandom number generator. The ServerHello message contains a similar nonce and the server’s choice of cipher suite. We specialize to the case of RSA key exchange with mutual authentication, an option that is enabled for higher-security deployments, for VPN-over-TLS, and other instances where the server needs assurance of the client’s identity. For these cipher suites, the server then sends a Certificate message with its certificate chain, a CertificateRequest message, and a ServerHelloDone message. The client checks the server certificate, generates a 48-byte premaster secret (PMS) and encrypts it to the server’s public key from the certificate. The PMS and padding formatting are shown above in Figure 4.

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

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

### D. Finding Randomness in TLS

The state recovery attack described in Section IV required 1996 bytes of output from the random number generator. Thus, for our cache side-channel attack to work at the protocol level, we needed to find places in the handshake where a single random number generator call would request enough output for an attacker to feasibly carry out state recovery. We evaluated the TLS protocol for potential sources of large or variable length randomness and settled on three possibilities: the ExtendedRandom TLS extension, RSASSA-PSS padding, and RSA PKCS#1 v1.5 padding.

**ExtendedRandom TLS Extension.** ExtendedRandom is a non-standard extension to TLS that was proposed to the IETF [25] to permit clients to request up to $2^{16} - 1$ bytes of randomness from the server. While our attacks (as well as those of Checkoway *et al.* [18] and Cohney *et al.* [19]) may have been able to make use of the increased output from the server’s generator to recover secret information, there are no known implementations with a functional implementation of this extension [25].

**RSA-PSS.** We evaluated whether the generation of the random salt for RSA-PSS signatures provided a viable attack vector. Under the PSS specification, for a message of $2^{14}$ bytes, the maximum salt length allowed is 2016 bytes, or 126 blocks of PRG output, sufficient for our state recovery attack. However, since RFC8446 [72] restricts the salt length when PSS is used in TLS1.2, an attacker in this context cannot observe enough encryptions from calls to the underlying PRG.

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

### E. Targeting TLS Clients

Unlike the attacks in [17–19], which compromise the server’s PRG, we compromise the state of the PRG used by the TLS client, since the client is the party that generates the encrypted key exchange message. However, similar to those works, we use the recovered state to predict future outputs of the PRG. In our case, this allows us to recover the client’s long-term authentication key.

**Attack Overview.** We assume the client connects to a malicious attacker-controlled server supporting TLS 1.0, 1.1, or 1.2 that uses RSA for key exchange, and that the client uses ECDSA for digital signatures. We also assume that the attacker is capable of running unprivileged code on the client. Next, since the RSA PKCS padding generation procedure requires the client to generate pseudorandom bytes, the attacker can use the cache leakage traces collected during the generation of the PKCS padding to recover the client’s PRG state via the method described in Section IV. With the client’s PRG state successfully recovered, the attacker predicts the subsequent PRG output and thus is able to compute the ECDSA nonce that the client generates in the course of producing the digital

signature for the CertificateVerify message. As outlined in Section VI-B, an attacker who knows the nonce used to generate an ECDSA signature can trivially recover the long-term private key used for client authentication, even if that key was generated in a secure manner. Recovering the signing key allows the attacker to impersonate the client. This may allow the attacker to access TLS-protected resources that are served only to an authenticated client. Our attack proceeds as follows:

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

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

We illustrate this challenge using OpenSSL 1.0.2, which we use as our baseline implementation for the nonce-recovery attacks described in this and following sections.

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

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

1. **Initial Padding Generation.** The output of the PRG is fed into an $n$-byte buffer to be used for PKCS#1 v1.5 padding, where $n$ is the length of padding required (in our case $n = 1996$). The state is updated twice, once before the bytes are generated and once after. State compromise occurs after the first call to update, but prior to the second.
2. **Padding Zero-Fill.** PKCS#1 v1.5 does not allow `0x00` bytes to be present in the random padding, so if there are $z$ `0x00` bytes present in the PKCS output buffer, OpenSSL makes at minimum $z$ more requests for output from the PRG, one for each byte. If any of these additional requests also result in a `0x00` byte, OpenSSL makes repeated requests to the PRG until the output is non-zero. The output from these requests is used to replace the null bytes in the padding to produce a valid non-null padding string under PKCS#1 v1.5. Within each request for random bytes, the PRG state is advanced twice. Both updates use the same underlying additional input.
3. **RAND_seed.** The ECDSA signing routine tries to reseed OpenSSL's RNG via `RAND_seed`. The SHA256 hash of the TLS handshake transcript is used as external entropy.
4. **RAND_add.** A call to `RAND_add` is made as part of `bnrand`, which is used to generate a random integer in a given range. Time in seconds is used as external entropy.
5. **GenNonce.** OpenSSL then generates the ECDSA nonce. Within the call to the CTR_DRBG generate function, the state is updated before the nonce value is finally produced.

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

**Causing a Large Number of Random Byte Generations.** To perform the attack, the attacker must observe side channel leakage during the generation of a large amount of randomness. Moreover, to recover the RNG's state, the attacker must learn the values of the victim-generated randomness. In our attack scenario, the attacker could cause a victim client to connect to the attack server using a malicious script served by an ad network on a website the user would otherwise normally visit. The attacker's malicious server is configured to support only RSA key exchange, and deliberately serves a 16534-bit RSA certificate, which is the maximum size that OpenSSL will support without throwing an error during the handshake.[3]

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

**The Problem of Padding Zero-Fill.** As noted above, to comply with the PKCS standard, there must be no `0x00` bytes in this padding. OpenSSL complies by first generating

[3]This is due to deliberate, hard-coded limits on message sizes that OpenSSL will accept, in the interest of preventing denial of service attacks [61, 63].

padding of the total length required, and then replacing each null byte with output from further calls to the PRG, each used to byte. To encrypt to the malicious server's large certificate, OpenSSL generates 1,996 bytes of output for padding used as per Figure 4. In expectation, a ciphertext will have eight such `0x00` bytes that need to be replaced.

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

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

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

### G. Implementation Choices and Nonce Recovery

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

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

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

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

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

**OpenSSL 1.1.1.** In OpenSSL 1.1.1 (the latest version at the time of writing) the maintainers rewrote much of the random number generation API. Due to the significant changes, this code was professionally audited twice [2, 71], both times finding only minor flaws with the PRG implementation. The implementation gathers additional input from a variety of sources and feeds it into an entropy pool. These include system event timing data, time, thread ids and output from the OS or hardware random number generators. Given this complexity, we did not estimate the entropy added in reseeding.

The ECDSA nonce generation mechanism in OpenSSL 1.1.1 was also improved. The nonce is generated from a hash of the private key, along with the transcript, and PRG output. The inclusion of secret data ensures that even if the PRG is compromised, the nonce cannot be recovered. Together, these measures preclude both state and nonce recovery.

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

### H. Evaluation

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

**The Victim.** For our victim TLS client, we used the sample TLS client code available in the OpenSSL documentation [60], configured to use mutual authentication and the nist_rng library with our choice of modeling parameters. We configured the client to authenticate using an ECDSA certificate with NIST P-256. For the ECDSA nonce, we used the raw PRG output, which matches the behavior of all implementations considered in Section VI-G, except OpenSSL 1.1.1.

**The Malicious TLS Server.** Our malicious server was the default OpenSSL tool, instrumented to dump TLS transcripts and ECDSA signatures to the filesystem, and configured to

| Target | Sources | Search Space | Reduced Space | CPU Time |
|---|---|---|---|---|
| **OpenSSL FIPS** | time, PID counter | $2^{24}$ | $2^{21}$ | 30 minutes |
| **NetBSD** | rdtsc | $2^{64}$ | $2^{46}$ | 2000 years |
| **FortiOS** | none | 0 | 0 | N/A |
| **Custom Params** | time | $2^{48}$ | $2^{43}$ | 200 years |

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

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

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

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

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

**Empirical Results.** Our attack succeeded against FortiOS in negligible time (following state recovery) and against OpenSSL FIPS after thirty minutes ($2^{21}$ work) using the hardware setup from Section V-B. For NetBSD and the custom parameters the search space was beyond our computational capabilities, and we terminated our search after approximately one hour of searching in both cases. We tabulate our results in Table I. While our experimental results are limited by our CPU's speed of $\approx 2^{22}$ elliptic curve scalar mutiplications per hour, Vanhoef and Ronen [83] achieve a rate of $2^{35}$ operations per hour using a commodity GPU. We therefore anticipate that using their setup, the custom parameters search ($2^{43}$ work) would be completed within two weeks. The attack on NetBSD ($2^{46}$ work) would likewise be completed in about 4 months on a single GPU.

**Handling AES-256.** To demonstrate key recovery under the constrained set of known ciphertexts available in the TLS setting of Section VI, we implemented our attack using AES-128. In Section VII, we handle AES-256 in the SGX setting.

## VII. Attacking Full Entropy Implementations

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

However, to achieve this, we require a stronger side-channel adversary, one who can observe the cache accesses during AES encryption at a high temporal resolution. Such an attacker can be achieved, for example, in the setting where the victim runs within an SGX enclave on a host with an attacker-controlled operating system. This scenario is within the threat model for SGX enclaves, and past research has demonstrated that it enables high resolution side-channel attacks [33, 80, 84].

We begin with background on SGX, cache attacks on SGX, and the SGX threat model (Section VII-A). We then present our novel differential cryptanalysis technique for exploiting side-channel information (Section VII-B). Finally, we evaluate our attack on an SGX port of the mbedTLS library (Section VII-D).

### A. Secure Enclave Technology

Intel Software Guard Extensions (SGX) [32] is an extension of the x86 instruction set that supports private regions of memory called *enclaves*. The contents of these enclaves cannot be read by any code running outside the enclave, including kernel and hypervisor code. This in theory allows a user-level process to protect its code and data from a highly privileged adversary, such as a malicious operating system or hypervisor.

**Cache Attacks on AES Inside SGX.** Although SGX protects the enclave from a malicious OS, it renders enclaved code *more* vulnerable to side channel attacks. Specifically, the cache attack of Section IV can only observe the overall access pattern over an entire encryption. However, when the victim runs in an SGX enclave, a malicious operating system can obtain much finer temporal resolution. This allows us to observe cache accesses after each of the 16 accesses to the AES T-tables in each of the encryption rounds [33, 80].

**Threat Model.** Following previous work [11, 55, 81, 89], in this section we assume a root-privileged attacker who controls the entire OS. This is in agreement with Intel SGX's threat model, where an enclave guarantees confidentiality and integrity, even against a malicious OS and hypervisor. Unlike

the attack described in Section VI, we do not assume that the enclaved TLS client is willing to connect to a malicious attacker-controlled server, or uses imperfect PRG reseeding.

### B. Differential Cryptanalysis of CTR_DRBG in the Presence of Side Channel Leakage

We provide the additional details about AES required for the differential attack. AES is a substitution-permutation cipher [8] that operates in a sequence of rounds on a 128-bit internal state $S$. Each round mixes the state and combines the mixed state with a round key. For a plaintext $x$, the initial state is $S_0 = x \oplus K_0$. Each consecutive round calculates $S_{j+1} = P(S_j) \oplus K_{j+1}$, where $P$ is the state mixing function and $K_j$ is the key for the $j^{\text{th}}$ round. For efficient software implementation, the mixing step is implemented using four T-tables. Each byte of the state selects one entry from a T-table and, since the T-table entries are 32 bits wide, each state bytes affects four consecutive bytes in the mixed state. For example, we can calculate the first four bytes of state $S_{j+1}$ by:

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

As before, our cache attack targets accesses to these T-tables. Because we cannot distinguish between entries in the same cache line, the cache leaks only the four most significant bits (MSBs) of each byte of the state in each round. Let $\langle\rangle_U$ denote setting the four least significant bits of each byte to zero, then the leakage on byte $k$ is $L_{j,k} = \langle S_{j,k}\rangle_U$. With a known plaintext $x$, we can use $L_{j,k}$ to recover the 4 MSBs of every byte of $K_0$ because $\langle K_{0,k}\rangle_U = \langle x_k\rangle_U \oplus L_{0,k}$.

Unfortunately, in our blind attack we do not know $x$. Consequently, we cannot learn information on $K_0$ from the leakage of the first round. Instead, we use the known difference between the plaintexts used in consecutive rounds of AES-CTR to recover the AES state. From the state, we can recover the keys, plaintexts, and ciphertexts. This is in close correspondence to the changes targeted in differential fault attacks [7]. We combine a closely related analysis with our side channel leakage to form the basis of our attack.

**Notation.** We use the following notation:

1) $T_0..T_3$ is the array of 4 AES T-Tables, where $T_i[j]$ is the value in location $j$ of Table $i$.
2) $\langle x\rangle_U$ denotes the value of $x$ with the lower four bits (nibble) in each byte set to 0.
3) $L_{i,j,k}$ is the value leaked from the cache attack for byte $k$ of round $j$ in trace $i$. The leaked value is only the 4 MSBs and the lower nibble is always 0.
4) $S_{i,j,k}$ is the real value of the state byte $k$ of round $j$ in trace $i$. $G_{i,j,k}$ is our current guess for this byte.
5) $R\Delta_{j,k}$ the value of the differential $S_{0,j,k} \oplus S_{1,j,k}$, and $\Delta_{j,k}$ is our current guess for this value.
6) $L\Delta_{j,k} = L_{0,j,k} \oplus L_{1,j,k}$ (lower nibble is always 0).
7) $K_{j,k}$ is the key value of byte $k$ of round $j$.

**Differential Analysis.** By analyzing the difference between the state of two encryptions, we can recover state information that is independent of the round keys. In AES-CTR, for two

**Algorithm 3** Find possible guesses for the last state 0 byte

```
1: function LASTSTATE0BYTE(L_{0,0,15}, Δ_{0,15}, LΔ_{1,0..3})
2:     GuessList0 ← Empty
3:     for Nibble ← 0 to 2^4 − 1 do
4:         G_{0,0,15} = L_{0,0,15} ⊕ Nibble
5:         Δ_{1,0..3} = T_3[G_{0,0,15}] ⊕ T_3[G_{0,0,15} ⊕ Δ_{0,15}]
6:         if ⟨Δ_{1,0..3}⟩_U = LΔ_{1,0..3} then
7:             GuessList0.append(G_{0,0,15}, Δ_{1,0..3})
8:     return GuessList0
```

consecutive plaintexts $x0$ and $x1$ we know that $x1 = x0 + 1$, so with probability (255/256) the two plaintexts only differ in the last byte by some value $\Delta_{ctr}$. As the state of round 0 is simply the plaintext XOR with $K_0$, the plaintext difference is preserved and $R\Delta_{0,15} = \Delta_{ctr}$. Using Equation (1) we get:

$$\begin{aligned}
S_{0,1,0..3} =& T_0[S_{0,0,0}] \oplus T_1[S_{0,0,5}] \oplus T_2[S_{0,0,10}] \\
& \oplus T_3[S_{0,0,15}] \oplus K_{i+1,0..3} \\
S_{1,1,0..3} =& T_0[S_{1,0,0}] \oplus T_1[S_{1,0,5}] \oplus T_2[S_{1,0,10}] \\
& \oplus T_3[S_{1,0,15}] \oplus K_{i+1,0..3} \\
=& T_0[S_{0,0,0}] \oplus T_1[S_{0,0,5}] \oplus T_2[S_{0,0,10}] \\
& \oplus T_3[S_{0,0,15} \oplus R\Delta_{0,15}] \oplus K_{i+1,0..3} \\
L\Delta_{1,0..3} =& L_{0,1,0..3} \oplus L_{1,1,0..3} = \langle S_{0,1,0..3} \oplus S_{1,1,0..3}\rangle_U \\
=& \langle T_3[S_{0,0,15}] \oplus T_3[S_{0,0,15} \oplus R\Delta_{0,15}]\rangle_U \quad (2)
\end{aligned}$$

As $\langle S_{0,0,15}\rangle_U = L_{0,0,15}$ we only need to try the 16 options for the lower four bits until we find a value that satisfies Equation (2) and recover $S_{0,0,15}$ (see Algorithm 3 ). As $R\Delta_{0,15}$ is unknown, we run Algorithm 3 with each possible value to retrieve the full set of candidates. However, as $R\Delta_{0,15} = x_0 \oplus x_0 + 1$ only eight candidates are possible. The full key and plaintext recovery procedures are described in Appendix A.

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

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

### C. Fine Grained Cache Attack

To generate the required traces, an attacker with OS level privileges (root) monitors cache access through Prime+Probe. The attack obtains fine-grained temporal resolution through a controlled-channel [89] attack. A controlled-channel attack involves disabling the present bit on the enclave's page tables, which by necessity are handled by the OS. By marking the page containing the T-Tables as not-present, the attacker forces an asynchronous enclave exit upon access to the table, thereby transferring control to the attacker controlled OS.

Since all of the T-Tables lie in the same page, the attacker must 'toggle' between the accesses by performing a controlled-channel attack on a page access that occurs in

between each T-Table access. We use the page containing the topmost frame of the stack for this, as the mbedTLS implementation must first read the index into the T-Table from the stack before each access.

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

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

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

**Related Attacks.** Roche *et al.* [73] also demonstrate a blind attack on AES. However, they consider a particularly powerful attacker who is able to generate arbitrary faults in the key schedule. Jaffe [38] attacked counter mode encryption with an unknown nonce, but required $2^{16}$ consecutive block encryptions. Ronen *et al.* [74] also showed a blind attack on counter mode encryption targeting the authentication MAC.

### D. Evaluation

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

Similar to [88], we demonstrate the viability of the differential attack against mbedtls-SGX [95], an SGX port of the widely-used mbedtls library. To the best of our knowledge, mbedtls-SGX is the only library currently available that features a function SGX-based HTTPS client.

**Attack Procedure.** We demonstrate an end-to-end attack on a connection between the TLS client and www.cia.gov, with all of the client's cryptographic operations taking place within the enclave. We first mount a Prime+Probe attack to recover the CTR_DRBG state used to generate the 256 bits of the ECDH ephemeral private key (a total of five AES256 encryptions of an incrementing counter). Using the recovered private key, we were able to calculate the premaster key and subsequently decrypt the HTTPS communication. The details of the side channel attack are left to Section VII-C.

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

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

TABLE II: **CMVP-Certified Uses of DRBG Designs.**

After recovering the PRG state, we recovered the TLS symmetric encryption keys and GCM IVs, and subsequently decryped the HTTPS request.

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

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

## VIII. Impact

In order to evaluate the impact of our findings, we scraped a public database of security certificates released under NIST's Cryptographic Module Validation Program (CMVP).

**Government Certification.** The CMVP allows vendors to certify that their cryptographic modules meet minimal requirements to sell to the United States and Canadian governments.

In order to comply with FIPS 140-2, implementations must use one of the PRGs described in SP 800-90A.

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

**Database Scraping.** We scraped a public facing database of CMVP certifications on May 13, 2019 to assess the potential impact of our findings. Our results are tabulated in Tables II–III. CTR_DRBG was the most popular design, supported by 67.8% of the implementations in the database. Of 2498 implementations present, 1694 (67.8%) supported CTR_DRBG. Of these 461 (25%) exclusively supported AES-128, 1163 (69%) supported AES-128 along with other ciphers, and 1227 (72%) supported AES-256. The CMVP database also permits modules to certify whether prediction resistance is enabled for the DRBG implementation. Of the 1694 total implementations

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

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

that supported CTR_DRBG, 66 provided no information about prediction resistance, 618 supported use of the DRBG in either mode with the default unspecified, 433 explicitly enabled prediction resistance, and 577 did not support prediction resistance. Among the CTR_DRBG implementations, 85 did not use a derivation function and 1137 did not support an alternate DRBG algorithm.

## IX. Discussion

**Limitations.** Our results rely on a victim's use of T-Table AES, which has long been known to leak information via side channels. However, as illustrated in this work T-Table AES is still used by many modern implementations. In the non-SGX setting, our TLS attack requires code execution on the client, and further succeeds only after thousands of handshakes. This potentially allows for detection of an ongoing attack. While we demonstrate our SGX attack against the only library that provides a working end-to-end example of an HTTPS client, the Intel-supported SGX-SSL cryptographic library [36] (which does not provide support for TLS) uses SGX's hardware-based `RDRAND` PRG and therefore is not vulnerable to a T-Table based attack.

**Countermeasures.** CTR_DRBG's flaws, both theoretical and practical, suggest that implementations need to take great care when choosing this design. Where FIPS compliance is required, HASH_DRBG and HMAC_DRBG give better security guarantees [86]. Where CTR_DRBG cannot be replaced, implementers should use AES hardware instructions, limit the quantity of data that can be requested in a single call, reseed frequently, and populate $addin$ with high quality entropy, to provide defense in depth against our attacks. In general, constant-time code should be used for all cryptographic applications, unless hardware support (e.g., AES-NI) is available.

**Mismatches Between Theory and Practice.** Significant effort has been dedicated to formalizing PRG security properties and designing provably secure constructions. However, theoretical analyses of many of the most commonly-used designs in practice (the Linux RNG [21], CTR_DRBG [86]) have found that these designs do not meet basic security properties such as robustness against state compromise. Unfortunately, implementers are often hesitant to adopt countermeasures without a concrete demonstration of vulnerability.

**The Fragility of ECDSA.** The fragility of DSA and ECDSA in the face of random number generation and implementation flaws has been repeatedly demonstrated in the literature [12, 91]. It is inevitable that a random number generation failure would compromise a single session or a signature, but DSA/ECDSA are particularly vulnerable to compromise of long-term secrets. Deterministic ECDSA, defined in RFC6979 [69], is the recommended countermeasure.

**Future of FIPS.** FIPS 140-3 is expected to contain requirements for side channel mitigations from the inclusion of NIST SP 800-140F, which has yet to be issued and becomes effective in September 2019. FIPS 140-2 CMVP certifications will be continue to be issued at least through 2021 [59]. This is a promising step towards widespread deployment of side-channel-resistant cryptography; however, it remains to be seen how improved requirements for certifying modules will feed back into the design and standardization of more secure primitives.

**Using `RDRAND` without a PRG.** Using the built-in CPU PRG to mitigate concerns with software PRGs is not a panacea. In several SGX ports we have reviewed (including Intel's official port for OpenSSL [36]) the software PRG was replaced with calls to the `RDRAND` instruction. While using the CPU's generator avoids software side channels, the existence of hard-to-discover bugs in PRGs integrated into CPUs [50, 87] mean this feature is better used as one of many sources of entropy for a provably secure software PRG.


## Acknowledgements

This work was supported by the National Science Foundation under grant no. CNS-1651344, by the ISF under grant number 1523/14, by gifts from Intel and AMD corporations, and by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-19-C-0531. Eyal Ronen is a member of CPIIS.


## References


[1] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *ACSAC*, 2016.

[2] J.-P. Aumasson and A. Vennard, *Audit of OpenSSL's randomness generation*, 2018. [Online]. Available: ostif.org/wp-content/uploads/2018/09/opensslrng-audit-report.pdf.

[3] J. Austen, *Pride and Prejudice*. 1813.

[4] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, "''Ooh aah... just a little bit" : A small amount of side channel can go a long way," in *CHES*, 2014.

[5] D. J. Bernstein, *Cache-timing attacks on AES*, 2005.

[6] ——, *Fast-key-erasure random-number generators*, 2017. [Online]. Available: blog.cr.yp.to/20170723-random.html.

[7] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *CRYPTO*, 1997.

[8] A. Biryukov, "Substitution–permutation (sp) network," in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Boston, MA: Springer US, 2011, pp. 1268–1268. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_619.

[9] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard pkcs# 1," in *CRYPTO*, 1998.

[10] J. Bonneau, *Robust final-round cache-trace attacks against AES*, IACR ePrint archive 2006/374, 2006.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *WOOT*, 2017.

[12] J. Breitner and N. Heninger, "Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies," in *FC*, 2019.

[13] S. Briongos, P. Malagón, J.-M. de Goyeneche, and J. Moya, "Cache misses and the recovery of the full AES 256 key," *Applied Sciences*, no. 5, 2019.

[14] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques," in *HPCA*, 2017.

[15] M. J. Campagna, "Security bounds for the NIST codebook-based deterministic random bit generator.," 2006.

[16] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, *A systematic evaluation of transient execution attacks and defenses*, arxiv:1811.05441, 2018.

[17] S. Checkoway, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, H. Shacham, and M. Fredrikson, "On the practical exploitability of dual EC in TLS implementations," in *USENIX Security*, 2014.

[18] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the Juniper dual EC incident," in *CCS*, 2016.

[19] S. N. Cohney, M. D. Green, and N. Heninger, "Practical state recovery attacks against legacy RNG implementations," in *CCS*.

[20] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *USENIX Security*, 2017.

[21] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs, "Security analysis of pseudo-random number generators with input," in *CCS*, 2013.

[22] C. Garcia and B. Brumley, "Constant-time callees with variable-time callers," in *USENIX Security*, 2017.

[23] GnuPG Project, *Gnupg*, 2019. [Online]. Available: www.gnupg.org.

[24] M. D. Green, *Twitter thread on openssl*. [Online]. Available: \url{https://twitter.com/matthew_d_green/status/1115013260783255558?s=12}.

[25] ——, *The strange story of "extended random"*, 2017. [Online]. Available: blog.cryptographyengineering.com/2017/12/19/the-strange-story-of-extended-random/.

[26] ——, *Wonk post: Chosen ciphertext security in public-key encryption (part 2)*, 2018. [Online]. Available: blog.cryptographyengineering.com/2018/07/20/wonk-post-chosen-ciphertext-security-in-public-key-encryption-part-2/.

[27] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, Gauss, and reload – a cache attack on the BLISS lattice-based signature scheme," in *CCS*, 2016.

[28] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015.

[29] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *DIMVA*, 2016.

[30] S. Gueron and Y. Lindell, "GCM-SIV," in *CCS*, 2015.

[31] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *IEEE S&P*, 2011.

[32] M. H., *Intel SGX for dummies (intel SGX design objectives)*, 2013. [Online]. Available: software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.

[33] M. Hähnel, W. Cui, M. Peinado, and T. Dresden, "High-resolution side channels for untrusted operating systems," in *USENIX ATC*, 2017.

[34] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices," in *USENIX Security*, 2012.

[35] S. Hirose, "Security analysis of DRBG using HMAC in NIST SP 800-90," in *WISA*, 2009.

[36] Intel, *Intel software guard extensions SSL*, 2017. [Online]. Available: github.com/intel/intel-sgx-ssl.

[37] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *RAID*, 2014.

[38] J. Jaffe, "A First-Order DPA Attack Against AES in Counter Mode with Unknown Initial Counter," in *CHES*, 2007.

[39] H. Jungheim, 2019. [Online]. Available: henric.org/random/#nistrng.

[40] B. Kaliski, "PKCS #1: RSA encryption version 1.5," RFC 2313, 1998.

[41] W. Kan, "Analysis of underlying assumptions in NIST DRBGs," 2007.

[42] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri, "MAGIC: Malicious aging in circuits/-cores," *TACO*, vol. 12, no. 1, p. 5, 2015.

[43] C. F. Kerry, A. Secretary, and C. R. Director, *FIPS pub 186-4: Digital signature standard (DSS)*, 2013.

[44] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits

in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, 2014.

[45] A. Klyubin, *Some SecureRandom thoughts*, 2013. [Online]. Available: android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html.

[46] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2018.

[47] A. Kurmus, N. Ioannou, N. Papandreou, and T. P. Parnell, "From random block corruption to privilege escalation: A filesystem attack vector for Rowhammer-like attacks," in *WOOT*, 2017.

[48] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[49] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security*, 2016.

[50] H.-T. Leung, *Redhat bug 1150286 - rdrand instruction fails after resume on AMD CPU*, 2019. [Online]. Available: bugzilla.kernel.org/show_bug.cgi?id=85911.

[51] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018.

[52] K. Michaelis, C. Meyer, and J. Schwenk, "Randomly failed! The state of randomness in current Java implementations," in *CT-RSA*, 2013.

[53] P. R. Mihir Bellare, *PSS: Provably secure encoding method for digital signatures*, 1998.

[54] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, *Fallout: Reading kernel writes from user space*, 2019.

[55] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[56] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA cryptography specifications version 2.2," RFC 8017, 2016.

[57] K. Mowery, S. Keelveedhi, and H. Shacham, "Are AES x86 cache timing attacks still feasible?" In *CCSW*, 2012.

[58] M. Neve and J.-P. Seifert, "Advances on Access-Driven Cache Attacks on AES," in *Selected Areas in Cryptography*, 2007.

[59] NIST, "Announcing issuance of federal information processing standard (FIPS) 140-3, security requirements for cryptographic modules," 2019.

[60] OpenSSL, *SSL/TLS Client*, 2018. [Online]. Available: wiki.openssl.org/index.php/SSL/TLS_Client.

[61] *Openssl software failure for RSA 16K modulus*, 2016. [Online]. Available: mta.openssl.org/pipermail/openssl-users/2016-July/004056.html.

[62] OpenSSL Software Foundation, *User Guide for the OpenSSL FIPS Object Module v2.0*, 2013.

[63] *[openssl.org #4063] re: Client hello longer than* $2^{14}$ *bytes are rejected*, 2015. [Online]. Available: mta.openssl.org/pipermail/openssl-dev/2015-September/002860.html.

[64] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and counter-measures: The case of AES," in *CT-RSA*, 2006.

[65] C. Percival, *Cache missing for fun and profit*, 2005.

[66] N. Perlroth, *Government announces steps to restore confidence on encryption standards*, 2013. [Online]. Available: bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards.

[67] P. Pessl, L. Groot Bruinderink, and Y. Yarom, "To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures," in *CCS*, 2017.

[68] R. Poddar, A. Datta, and C. Rebeiro, "A cache trace attack on CAMELLIA," in *InfoSecHiComNet*, 2011.

[69] T. Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," RFC 6979, 2013.

[70] T. O. Project, *OpenSSL: The open source toolkit for SSL/TLS*, 2003.

[71] Quarkslab SAS, *Openssl security assessment*, 2019. [Online]. Available: ostif.org/wp-content/uploads/2019/01/18-04-720-REP_v1.2.pdf.

[72] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC 8446, 2018.

[73] T. Roche, V. Lomné, and K. Khalfallah, "Combined fault and side-channel attack on protected implementations of AES," in *CARDIS*, 2011.

[74] E. Ronen, A. Shamir, A. O. Weingarten, and C. Oflynn, "IoT goes nuclear: Creating a Zigbee chain reaction," in *IEEE S&P*, 2018.

[75] S. Ruhault, "SoK: Security models for pseudo-random number generators," *FSE*, 2017.

[76] T. Shrimpton and R. S. Terashima, "Salvaging weak security bounds for blockcipher-based constructions," in *ASIACRYPT*, 2016.

[77] D. Shumow and N. Ferguson, "On the possibility of a back door in the NIST sp800-90 dual EC PRNG," in *CRYPTO*, 2007.

[78] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned AES T-Tables," in *Constructive Side-Channel Analysis and Secure Design*, 2013.

[79] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *Journal of Cryptology*, no. 1, 2010.

[80] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step," in *SysTEX*, 2017.

[81] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution,” in *USENIX Sec*, 2018.
[82] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” in *IEEE S&P*, 2019.
[83] M. Vanhoef and E. Ronen, “Dragonblood: A security analysis of wpa3’s sae handshake.,” *eprint*, 2019.
[84] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, “Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX,” in *CCS*, 2017.
[85] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*, 2018.
[86] J. Woodage and D. Shumow, “An Analysis of the NIST SP 800-90A Standard.,” in *EUROCRYPT*, 2019.
[87] Wtdrog, *Systemd Issue #11810 - Can’t suspend again after suspending one time*, 2019. [Online]. Available: github.com/systemd/systemd/issues/11810.
[88] Y. Xiao, M. Li, S. Chen, and Y. Zhang, “Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves,” in *CCS*, 2017.
[89] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *IEEE S&P*, 2015.
[90] M. Yan, C. Fletcher, and J. Torrellas, *Cache telepathy: Leveraging shared resource attacks to learn DNN architectures*, arxiv:1808.04761, 2018.
[91] Y. Yarom and N. Benger, *Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack*, IACR ePrint archive 2014/140, 2014.
[92] Y. Yarom and K. Falkner, “textsc Flush+Reload : a high resolution, low noise, L3 cache side-channel attack,” in *USENIX Security*, 2014.
[93] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, “Verified correctness and security of mbedTLS HMAC-DRBG,” in *CCS*, 2017.
[94] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, “When private keys are public,” in *IMC*, 2009.
[95] F. Zhang, *mbedtls-SGX*, 2018. [Online]. Available: github.com/bl4ck5un/mbedtls-SGX.
[96] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, “TruSpy: Cache side-channel information leakage from the secure world on ARM devices,” *ePrint*, 2016.
[97] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant side-channel attacks in PaaS clouds,” in *CCS*, 2014.

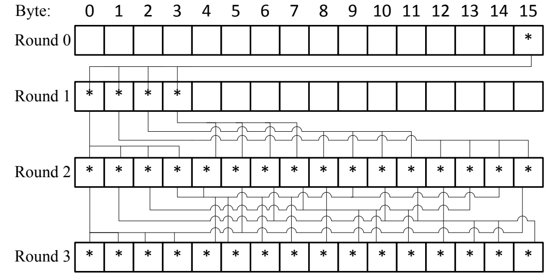


Fig. 5: Single byte differential propagation in AES state

**Algorithm 4** Find possible guesses for byte 0 of Round 1

```
1: function BYTE0ROUND1(L_{0,1,0}, Δ_{1,0}, LΔ_{2,0..3})
2:     GuessList1 ← Empty
3:     for Nibble ← 0 to 2^4 − 1 do
4:         G_{0,1,0} = L_{0,1,0} ⊕ Nibble
5:         Δ_{2,0..3} = T_0[G_{0,1,0}] ⊕ T_0[G_{0,1,0} ⊕ Δ_{1,0}]
6:         if ⟨Δ_{2,0..3}⟩_U = LΔ_{2,0..3} then
7:             GuessList1.append(G_{0,1,0}, Δ_{2,0..3})
8:     return GuessList1
```

## APPENDIX A
## THE FULL DIFFERENTIAL CRYPTANALYSIS

**Differential Propagation.** In a differential attack we can only recover state bytes that differ in the two encryptions. Our attacks thus follows the “differential propagation” in the AES rounds as shown in Figure 5. This will allow us to recover one byte of state of round 0, 4 bytes of the state of round 1, and the entire states from round 2 and above.

**From State to Key and Plaintext Recovery.** Assuming we were able to recover the full values of the states in rounds $j$ and $j+1$, we can now recover the key for round $j+1$:

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

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

**Iterative State Guess Elimination.** In the beginning of step $j$ of our attack we have one or more possible guesses for the values of the state bytes of round $j$. For each guess we enumerate all possible guesses for state bytes in round $j+1$, and efficiently eliminating guesses that does not satisfy the above equations for the “differential propagation”. The remaining guess are used as input for the next step of the attack. When we have a guess for the state of two full rounds we can try to recover the plaintext of the traces and verify that they are indeed a part of an incriminating counter.

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

**The Full Attack.** As we have seen we can retrieve GuessList0 that contains all possible guess for $G_{0,0,15}$ and $\Delta_{1,0..3}$ using Algorithm 3. For each guess in GuessList0 we

**Algorithm 5** Find possible guesses for byte 0, 5, 10 and 15 of round 2

```
1: function BYTE0-5-10-15-ROUND2(L_{0,2,(0,5,10,15)}, Δ_{2,(0,5,10,15)}, LΔ_{3,0..3})
2:     GuessList2 ← Empty
3:     IndxList ← [0,5,10,15]
4:     for Guess ← 0 to 2^16 − 1 do
5:         Δ_{3,0..3} = 0
6:         for i ← 0 to 3 do
7:             Nibble = (Guess >> (i ∗ 4))&0xf
8:             G_{0,2,IndxList[i]} = L_{0,2,IndxList[i]} ⊕ Nibble
9:             Δ_{3,0..3} ⊕= T_i[G_{0,2,IndxList[i]}]
10:            Δ_{3,0..3} ⊕= T_i[G_{0,2,IndxList[i]} ⊕ Δ_{2,IndxList[i]}]
11:            if ⟨Δ_{3,0..3}⟩_U = LΔ_{3,0..3} then
12:                GuessList2.append(G_{0,2,(0,5,10,15)}, Δ_{3,0..3})
13:    return GuessList2
```

now try to recover $4$ bytes from round $1$ using a similar method. As each of the $4$ bytes affect different $4$ bytes in round 2, we run the same algorithm as in step $1$ using different values. In Algorithm 4 we show how to find the possible guesses for $G_{0,1,0}$. A similar function will find the possible guesses for $G_{0,1,1}$, $G_{0,1,2}$ and $G_{0,1,3}$. As we may have more than one guess for each byte value, the full guess list GuessList1 is all the possible combinations of the different guesses for each of the bytes.

In the third phase of our attack, we try to generate all of the possible guesses for the entire state of round 2. Due to the "Shift Row" transformation of AES, the value of each of the $4$ bytes in round $1$ affect the values of distinct $4$ bytes in round $2$ (see Figure 5). The same guessing logic as before allows us to create the new guess list (see Algorithm 5 for example). The guess list GuessList2 is created from all the possible combinations of the guess for each $4$ byte group (this is done separately for each guess in GuessList1). We can repeat the same process to use GuessList2 to create the guess list GuessList3 for the state of round $3$ (and in the case of AES 256 continue another round to get GuessList4).