Surreptitiously Weakening Cryptographic Systems

Bruce Schneier[1] Matthew Fredrikson[2] Tadayoshi Kohno[3] Thomas Ristenpart[2]

[1] *Co3 Systems* [2] *University of Wisconsin* [3] *University of Washington*

February 9, 2015

Abstract

Revelations over the past couple of years highlight the importance of understanding malicious and surreptitious weakening of cryptographic systems. We provide an overview of this domain, using a number of historical examples to drive development of a weaknesses taxonomy. This allows comparing different approaches to sabotage. We categorize a broader set of potential avenues for weakening systems using this taxonomy, and discuss what future research is needed to provide sabotage-resilient cryptography.

1 Introduction

Cryptography is critical to modern information security. While it is undeniably difficult to design and implement secure cryptographic systems, the underlying mathematics of cryptography gives the defender an unfair advantage. Broadly speaking, each additional key bit increases the work of encryption and decryption linearly, while increasing the cryptanalysis work exponentially. Faced with this unbalanced work factor, attackers have understandably looked for other means to defeat cryptographic protections. Undermining the cryptography – by this we mean deliberately inserting errors or vulnerabilities designed to make cryptanalysis easier – is one promising avenue for actors seeking access to data, communications, or computing systems.

While academics begun studying sabotage of crypto systems before their deployment almost two decades ago [73, 80], the recent allegations that the United States government has subverted various cryptographic standards reemphasizes the need to understand avenues for, and defenses against, the deliberate weakening of cryptographic algorithms. This topic not only affects those worried about US government surveillance; clearly there are many nation-states and non-nation-state organization with incentives and expertise suitable to perform this type of sabotage.

This paper is an attempt to survey the field of surreptitiously weakening cryptographic systems. We first overview a number of historical cases of weaknesses built into cryptographic systems. These range from what were very likely to be benign bugs, which point to techniques that could be coopted by saboteurs, to explicit attempts to surreptitiously subvert systems. While in preparing this paper we investigated a larger number of weaknesses, we focused on a handful of examples for their relevance and breadth across the various approaches for undermining security We classify these historical examples according to a taxonomy that we develop. The taxonomy characterizes weaknesses by their "features," at least according to the point of view of a saboteur. This also allows us to compare weaknesses across a variety of axes. While our assessments are primarily qualitative, we believe it nevertheless sheds light on how to think about sabotage.

We then explore hypothetical ways in which sabotage may arise, across both designs and implementations, and in two case studies of TLS and BitLocker. We end with a discussion of what future directions may be profitable for those seeking to defend against cryptographic sabotage.

2 Threat Model and Terminology

The setting and players. We are concerned with mechanisms that an attacker can embed in instances of a cryptosystem to subvert some of its intended security properties. We call such a mechanism the *weakness*, and the party that is responsible for embedding it the *saboteur*. In some cases, we need to distinguish between

the saboteur and the *attacker(s)* that, at a later time, make malicious use of the weakness to target one or more legitimate *user(s)* of the system. Finally, the *defenders* are those responsible for finding or preventing weaknesses in instances of the targeted cryptosystem.

We make this terminology concrete with two examples. The first is the dual elliptic curve, or Dual EC, pseudorandom number generator (PRNG), standardized by NIST [16,58,69]. The built-in weakness lies with the choice of elliptic curve points that parameterize the algorithm: someone who chooses these points can establish a trapdoor that enables predicting future outputs of the generator given one block of output. The saboteur here is the party that generates those default curve points, allegedly the National Security Agency (NSA); the defender is NIST; the victims include any user of software relying on the generator including some TLS libraries such as RSA's BSAFE; and the attacker is the party that uses the trapdoor information to exploit the weakness against a (say) TLS session. We discuss this backdoor in more detail in Section 5.

Saboteur	Adds weakness to crypto system
Victim	User of the weakened crypto system
Attacker	Exploits weakness to attack victim
Defender	Designers of crypto systems

Figure 1: Terminology and players.

Another example, also related to random number generation, is a weakness introduced into a version of OpenSSL distributed with Debian-based operating systems from September 2006 to May 2008 [1]. The weakness, which appears to have been accidentally added, stemmed from a reduction in source entropy for the PRNG to only 15 bits, making it easy to brute-force guess outputs of the PRNG. This had widespread repercussions for Debian users, including secret key exposure. Here the saboteur was a Debian package maintainer who removed certain lines from the OpenSSL source when he discovered that his debugging tools flagged them as probable bugs. Note that although we use the term saboteur for consistency, in this case he appeared to have introduced the weakness by accident and was not working with any of the attackers that eventually were able to exploit the vulnerability, i.e., our use of the term saboteur is meant to describe a role, not an intent. We will discuss this vulnerability further in Section 5.

On surreptitiousness. We emphasize that we focus on *surreptitious* weakenings of cryptographic algorithms. The alleged backdoor in Dual EC discussed above was claimed to be unknown to NIST at the time of standardization [61], and certainly implementors of the standard were mislead by implicit and explicit claims of security. The OpenSSL vulnerability likewise went unnoticed for several years. In contrast, an example of a weakening not passing the surreptitiousness bar is that of the Data Encryption Standard (DES) block cipher. As originally conceived by IBM in the 1970's, DES used 64-bit keys. Prior to standardization, the National Security Agency (NSA) stipulated a shorter 48-bit key size, and a compromise was reached with 56-bit keys [44]. Although 56-bit keys were at the time strong enough to prevent attacks by most parties, many have hypothesized that this was an effective weakness exploitable by the NSA, who had access to superior cryptanalytic techniques and computation resources. While this is an example of a cryptographic weakening, it is not a surreptitious one, as the weakening from 64 to 56 bits was obvious to all at the time of DES's standardization and use (c.f., [28]).

On non-cryptographic sabotage. We also consider as explicitly out-of-scope backdoors or other mechanisms that undermine non-cryptographic security mechanisms. A fantastic example of such sabotage is the attempt to insert a backdoor into the Linux kernel in 2003 [35]. Some unknown attacker had compromised a source code repository for the Debian Linux kernel and made changes to the `wait` function so that any process could call it with specially crafted arguments and obtain root privilege. While this malicious change to the source code was caught before it could affect deployments, it serves as a good example of how software can be subverted. These threats are clearly important, and may even be more damaging than cryptographic weaknesses in some contexts, but we will not consider them further.

Saboteur goals. There are many conceivable outcomes of introducing a weakness into a cryptosystem, so rather than attempt to enumerate many of them, we will list some of the immediate technical consequences that a saboteur might desire:

- *Key disclosure:* One of the most attractive goals is disclosure of secret key data. Knowledge of a user's secret key enables a wide range of subsequent attacks.

- *Plaintext disclosure:* Learning the plaintext of a user's ciphertext is the canonical goal of eavesdroppers.
- *Forgery or impersonation:* In many scenarios, the ability to forge message or signature data under a user's secret key is desirable. Often this goes hand-in-hand with impersonation or authentication attacks, as in the case of man-in-the-middle attacks on encrypted streams.
- *Bootstrapping:* Certain attacks serve primarily to enable other attacks. For example, version/cipher rollback attacks [76] allow the adversary to "trick" the user into using a cryptosystem vulnerable to a secondary attack.

A specific weakness can enable several of these goals. For example, the Dual EC weakness could be considered to be a type of bootstrapping attack that enables all of the other goals. However, the goals can still be exclusive. For example, a symmetric cipher deliberately designed to be vulnerable to certain types of cryptanalysis may not necessarily yield secret key data or enable forgery.

3 Prior Research on Cryptographic Backdoors

We review several lines of work that introduce frameworks for reasoning about weakening cryptographic implementations or algorithms. Particular historical instances of weak cryptosystems will be discussed later, in Section 5.

Subliminal channels. Simmons undertook early work on subliminal channels, which are channels designed to covertly send information out of a cryptosystem. Note that these differ from steganographic channels in one important regard: even if an adversary is aware of the existence of a subliminal channel, he cannot decode any messages transmitted over it, and in some cases may not be able to detect it. Generally, subliminal channels exploit randomness used by the cryptosystem, using specially-crafted acceptance/rejection criteria on the random bits delivered to the cryptosystem to leak private information in its output. Work in subliminal channels is not concerned with source-level concealment, assuming only black-box interactions without timing information. For this reason, it is mainly of theoretical interest, although some of the results have serious implications for hardware-based implementations of cryptography.

This work was first done in the context of nuclear non-proliferation treaty enforcement between the United States and Soviet Union (see [73] for an extensive retrospective account). Briefly, the NSA wanted to use a cryptosystem designed by the Soviet Union to produce proofs of treaty compliance. Simmons and others pointed out the dangers of doing so, arguing that a maliciously-designed system might leak sensitive information, such as the location of active missile silos, back to the Soviet Union. Simmons showed that the Rabin cipher had a peculiar property that could be used to thwart the goals of the treaty enforcement program. Namely, for every plaintext and key, there exists a pair of distinguishable (to the adversary) ciphertexts that decrypt to the original plaintext using the given key—to leak a single bit, the system only needs a way of systematically choosing which ciphertext represents each bit value. This was essentially an existence proof of a one-bit subliminal channel in a modern cryptosystem.

Subsequently, Simmons studied the *prisoner's problem* and a number of associated subliminal channels in digital signature algorithms [70]. In the prisoner's problem, two prisoners are allowed to communicate by sending messages through the warden. Their goal is to coordinate an escape plan without being detected by the warden, while the warden's goal is to deceive either prisoner into accepting a fraudulent message. Thus, they make use of a signature protocol, which Simmons shows is sufficient to allow a subliminal channel that solves the problem. He first showed the feasibility of subliminal channels in signature protocols under the assumption that both parties know the private signing key. In a series of later findings, he showed that both the ElGamal and Schnorr signature algorithms contained subliminal channels without assuming a shared signing key [71], as well as a narrow-band channel in DSA that could be used to leak the signing key [72], thus bootstrapping his earlier channel. The channel assumes shared knowledge of a set of large primes, and leaks a chosen set of bits through the Legendre symbol of the signature's quotient with each prime; the signature's random per-message value is chosen to ensure the quotient produces this effect for each bit.

Implementation substitution attacks. A line of work initiated by Yung and Young [80] explored settings in which a saboteur replaces an implementation with a backdoored one, an area they refer to as

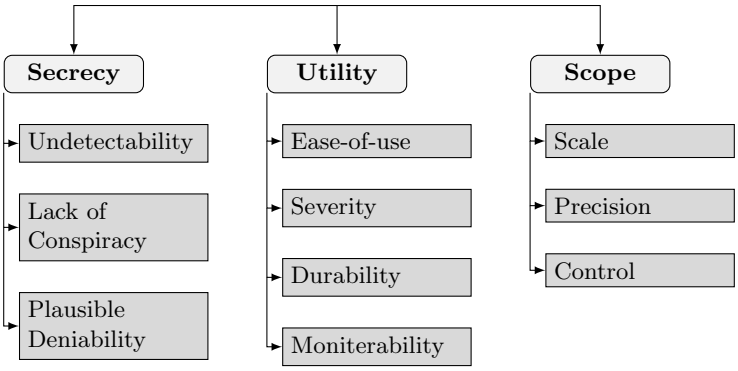

Figure 2: Taxonomy for comparing cryptographic weaknesses. We will generally use qualitative assessments of low, medium, and high for each property. For all properties, higher is better for the saboteur and worse for the defender.

kleptography. In its original conception, they consider a saboteur who designs an cryptographic algorithm whose outputs are computationally indistinguishable from the outputs of an unmodified trusted algorithm. The saboteur's algorithm should leak private key data through the output of the system using the same principles as Simmon's earlier subliminal channels. They call such a subversion a SETUP attack ("Secretly Embedded Trapdoor with Universal Protection"). Note that as with subliminal channels, these attacks do not try to hide their presence in a particular implementation; from a practical perspective, they are relevant mainly to closed hardware implementations. Young and Yung presented SETUP attacks on RSA key generation, by encoding one of the private primes in the upper-order bits of the public modulus. Security and forward-secrecy are obtained by first encrypting the prime with the attacker's public key. They later designed SETUP attacks for Diffie-Hellman key exchange [81], DSA, ElGamal encryption, ElGamal signatures, and Schnorr signatures [82]. On a similar note, Blaze, Feigenbaum, and Leize describe master-key cryptosystems [11], which are simply trapdoor encryption in the same spirit as the SETUP attacks.

Bellare, Paterson, and Rogaway [6] explore SETUP attacks for case of symmetric encryption, and rebrand SETUP attacks as algorithmic substitution attacks (ASAs). They give SETUP attacks against symmetric encryption, but also seek countermeasures, in particular arguing that a symmetric encryption scheme is secure against sabotage if no attacker, even given a secret trapdoor, can distinguish between ciphertexts generated by the trusted reference algorithm versus ones generated by a backdoored version of it. They argue that deterministic, stateful schemes can be shown to meet this latter notion. However, it is important to note that this result is only meaningful if the algorithm underlying the reference implementation is assumed to be free of backdoors; this is not the case in several of the historical examples discussed in Section 5. More generally, the SETUP and ASA frameworks do not capture all relevant aspects of a cryptographic weakness, nor potential routes for a saboteur to achieve one. We enumerate some of these broader dimensions in the following section and go into detail on methodologies for sabotage later.

4 Comparing Weaknesses: A Taxonomy

Weaknesses in cryptosystems can be characterized by properties that characterize their effectiveness in various settings. For example, the weakness in Dual EC can only be exploited by the holder of the trapdoor information, giving the saboteur some ability to control which attackers are able to exploit it. On the other hand, the weakness in the Debian PRNG does not rely on any secret information, and could be rediscovered and used by anyone. In this section, we describe a number of properties that are useful when comparing the merits and shortcomings of cryptosystem weaknesses. They are organized along three categories: secrecy, utility, and scope. Each weakness attribute is *saboteur-positive*, meaning that higher is better for saboteurs and lower is worse for saboteurs. A summary of our taxonomy is given in Figure 2. It should be noted that the assessments we make for particular examples are necessarily subjective, but this should not detract from the purpose of the taxonomy.

Secrecy: In most cases, the saboteur has an interest in keeping the presence of a weakness unknown or uncertain. In its purest sense, secrecy is a measure of detectability. That is, the degree of difficulty faced by the defender in recognizing the presence of a weakness in an instance of the cryptosystem. Secrecy also relates to a number of more subtle properties that may be relevant for certain weaknesses depending on the context in which they are introduced and managed.

- *Undetectability* measures the difficulty with which the weakness can be discovered. This could be quantified by the type of work and expense of uncovering the backdoor, for example whether it requires black-box testing, source-code review, or even reverse engineering or new cryptographic insight.
- *Lack of conspiracy* measures the (lack of) complexity of embedding a backdoor at the scale intended by the saboteur (e.g., a single instance used by a particular target, or every instance of a standardized specification), and subsequently maintaining its undetectability. This could be quantified by the number of people — possibly at different organizations — who must have had knowledge of a weakness prior to and during its lifespan, as well as the resources needed to maintain it.
- *Plausible deniability* measures the extent to which a weakness could be considered *benign* or *unintentional* should it be discovered. For example, if it were discovered that the OpenSSL PRNG weakness was planned, we would call it plausibly-deniable because its technical root cause could follow from the honest behavior of an engineer lacking cryptographic expertise.

While objectively quantifying secrecy can be difficult, oftentimes anecdotal evidence can help. For example, once a weakness is found, we can determine how long the weakness went unnoticed to get a general sense of secrecy.

Utility: Utility is characterized by the amount of work needed to make effective use of a weakness, and the conditions under which it may be used, characterize its utility. Depending on the attacker's goals and context, there are several ways to evaluate the utility of a weakness.

- *Ease of use* characterizes the computational and logistical difficulty of mounting an attack using the weakness. For example, the Dual EC backdoor can be tricky to exploit in certain situations [16], whereas the OpenSSL PRNG weakness provided attackers with a small public/private SSH key pair table for mounting key disclosure attacks [1, 79].
- *Severity* measures the degree to which the weakness directly enables the attacker's goal. For example, a weakened cipher may leak certain features of the plaintext without giving access to all of its bits, or only a fraction of the key bits making brute-force attacks easier but not trivial. Certain oracle attacks may leak progressive amounts of plaintext or key data [75], but only after extensive interaction on the part of the attacker, giving the user an opportunity to recognize and prevent further leakage.
- *Durability* measures the difficulty of removing the weakness or preventing its use, once it has been disclosed to the defender. The Debian OpenSSL PRNG weakness would classify as moderately durable, as while the fix is easy it took a relatively long time for systems to deploy it [79]. Fixing the Dual EC weakness required deprecating the algorithm, though most implementations could switch to another algorithm via a configuration change.
- *Moniterability* measures the ability of the saboteur to detect exploitation of the weakness by attackers. (Note that this is different than being able to detect that a victim is vulnerable to an attacker.) Monitering is exclusive if only the saboteur can detect use of the weakness (e.g., due to needing knowledge of a secret trapdoor), and otherwise is permissive. Weaknesses with high moniterability would be particularly useful in situations where the saboteur cannot limit collateral damage (see below), but may be able to mitigate it once others begin to exploit it (e.g., by anonymously reporting the weakness to vendors).

Utility can be difficult to quantify because of the various meanings of the word in different contexts. However, it may be possible to quantify its sub-properties with more precision to get a better sense of this measure: how many CPU-hours are required to exploit the weakness, how long was the weakness exploitable, and how much did it cost to fix.

Scope: Weaknesses differ in how many users are potentially affected by their presence. A weakness in a government standard will affect everyone who subscribes to the standard, whereas a weakness in a particular

version of a library or implementation of a protocol will affect a smaller population. The scope of a weakness may be an intentional feature designed by the saboteur, or an unintentional side effect arising from an opportunistic weakening. Scope also relates to the degree to which an attacker can manage the effects of his activities:

- *Scale* measures the number of potential victims. Dragnet surveillance is enabled by a weakness that turns a large fraction of the population into victims. Other weaknesses can only be used against a particular target.
- *Precision* measures the granularity at which a saboteur can embed the weakness. Given a high-precision weakness, a saboteur can weaken only target victims' systems. A low-precision weakness forces the saboteur to weaken a large number of users' systems in pursuit of a target. An example of the latter is a weakness in some mandatory standard, an example of the former is a change to a single user's implementation.
- *Control* measures the degree to which the saboteur can limit whom can exploit a weakness as an attacker, even after it becomes public. For example, Dual EC is only exploitable by someone who knows the constant relating the NIST curves used to generate random bits, and an outsider must solve an instance of the elliptic curve discrete logarithm problem (ECDLP) to recover it. These properties give Dual EC a high degree of control. In comparison, when a weakness has low control, such as with something like the Debian OpenSSL or Heartbleed bugs, there can be significant collateral damage as arbitrary parties can exploit the weakness.

Although obtaining reliable measurements can be expensive, scope is relatively easy to quantify. Possible issues arise when the weakness exhibits randomness in its utility. For example, it may be difficult to accurately characterize the scope of a key generator that occasionally produces weak keys.

5 Historical Examples of Sabotage

We now review several known or alleged examples of sabotage, which serve to both exercise our taxonomy of the last section as well as educate our breakdown of ways by which sabotage is likely to occur (Section 6).

Lotus Notes. In the 1990s US companies were required to abide by cryptographic export restrictions for products marketed internationally. This included requiring only 40-bit key strength for symmetric keys. Many companies simply used symmetric encryption schemes with 40-bit keys, while Lotus Notes 4.0 did something different for internationally licensed versions of its product. Anytime a plaintext was encrypted, 24 bits of the used 64-bit symmetric key was separately encrypted (along with sufficient random padding) under an RSA public key belonging to the NSA. The new ciphertext was stored with the conventional ciphertext. To attempt to prevent tampering, Lotus Notes would refuse to decrypt properly a ciphertext with missing or incorrect NSA ciphertext. (This required encrypting the random padding with the actual 64-bit key, and recomputing the NSA ciphertext using it and the 24-bits of key.) This whole technology was referred to as Differential Workfactor Cryptography, and advertised as a mechanism by which Lotus Notes met export requirements while preventing non-NSA actors from breaking the encryption with less than 64-bits worth of work. The mechanism was designed in whole or part by engineers at Lotus Notes [46, 47].

The Lotus Notes example is arguably not surreptitious. The developers did not hide the fact and even seemingly argued that it was a competitive advantage over other encryption tools that did not achieve full key strength (against outsider attacks). Users of the application may have been completely unaware of the weakness, however, and so we include it as an example.

This backdoor has high detectability, as the feature was in technical documentation, though, as said, customers may nevertheless not have realized that the backdoor was included. It has high conspiracy, requiring the cooperation of engineers and management at Lotus Notes as well as the US government, which provided the public key used. It is very easy to use, as the holder of the saboteur can give the attacker the secret key enabling trivially recovering 24-bits of the key and, from there, the attacker can brute-force the remaining 40-bits. The backdoor seemingly lasted for the lifetime of Lotus Notes 4, as references to this mechanism are no longer present in the manual for version 5, and it has reasonably high durability as one could not easily remove the weakness from this proprietary product. As the weakness can be exploited

passively (given knowledge of the secret key), it is difficult for anyone to monitor exploitation of the backdoor. The scale was very large as Lotus Notes was a widely used product at the time, for example there were purportedly 400,000 to 500,000 users in Sweden alone [53]. The precision is moderate as all encryptions generated by the one program included the vulnerability, but the control is high as it requires access to the special secret key.

Dual EC_DRBG. The FIPS standard for pseudorandom number generators includes one called Dual elliptic-curve deterministic random byte generator. This EC_DRBG includes elliptic curve points as parameters for the system. It was observed in 2005 by Ferguson and Shumow [69] that whoever picked these curve points could do so in a way that enables them to, later, predict the randomness generated by the system. Several Snowden revelations later suggested that the NSA chose these parameters on behalf of NIST and pushed for inclusion of this algorithm in the FIPS standard. Allegedly, the RSA corporation was also paid a significant sum by the US government to make EC_DRBG the default algorithm used by its BSAFE library [56] (used, for example, to implement TLS in a variety of settings) . To use the weakness the attacker must obtain some number of bytes of output by it, after which future outputs can be predicted. This would be possible if, for example, TLS nonces that are sent in the clear are taken from EC_DRBG output and later secret data such as session keys are also derived from EC_DRBG. (See Section 7 for more details regarding TLS.) EC_DRBG is currently being deprecated.

We view this weakness as having moderate detectability — while trained cryptographers eventually discovered the possibility of the bug, the majority of people seem to have not realized that the backdoor existed. (Both the NIST standardizers and RSA claim to have not known there was a backdoor.) It has high conspiracy, as it required getting the standard included in FIPS, as well as getting companies to use it as the default. It has low plausible deniability, as once revealed there is no way to claim benign cause for knowing the trapdoor.

It is moderately easy to use: it highly depends on implementation details whether the trapdoor is usable [16]. Like the Lotus Notes weakness, the moniterability of this is low as it can be exploited passively. The durability is moderate as, while there are easy, secure alternatives (even in the FIPS standard specifying EC_DRBG), standardized algorithms are notoriously difficult to deprecate. Being a standard, its scale is that of adoption as the default usage. And precision is low as all users of the standard are subject to the weakness. It has high control, since without the trapdoor information no one else can exploit the weakness.

Debian OpenSSL PRNG. In 2006, a Debian developer commented out two lines of source code in the Debian OpenSSL implementation. These code lines were causing a warning by valgrind (a security tool) due to non-standard memory manipulations. Unfortunately, they were also the lines of code responsible for adding the bulk of fresh entropy to the OpenSSL random number generator (RNG). This meant that TLS or SSH host keys generated by affected systems (essentially all Debian systems) took on only one of 32,767 key pairs. This is a critical cryptographic vulnerability, enabling users to impersonate the affected systems, login to systems that had vulnerable public-keys installed, or decrypt TLS and SSH communications. While there is no reason to expect that this was anything but an honest mistake by the developer, we nevertheless consider it as an example of sabotage for this paper. Recall that we use the term to describe an actor or action, not an intent.

We view this weakness as having high undetectability (it took 2 years to detect despite the availability of source code), low conspiracy (requiring only one developer's changes to be accepted), and high plausible deniability. It is very easy to use, since it is easy to detect if a system is affected and, in this case, mounting attacks can be done by precomputing all of the possible secret keys. The durability is low, as a simple patch removes the weakness. (Patches may take a long time to get deployed, of course, see [79].) The severity was high, since it enables full system compromise. It only affects one distribution, albeit with wide install base, which suggests medium precision. The moniterability is moderate, as one could setup honey pots to detect indiscriminate use of the vulnerability or could inspect network traces to detect use of a vulnerable key to log into a system. (Though this would look indistinguishable from legitimate use, in terms of the cryptographic portions of the transcript.) Finally, it has low control, as, once discovered, anyone can exploit the weakness.

Heartbleed. In April of 2014, a buffer overread vulnerability in OpenSSL's implementation of the TLS and DTLS heartbeat extension was publicly disclosed. An attacker can exploit the vulnerability by sending

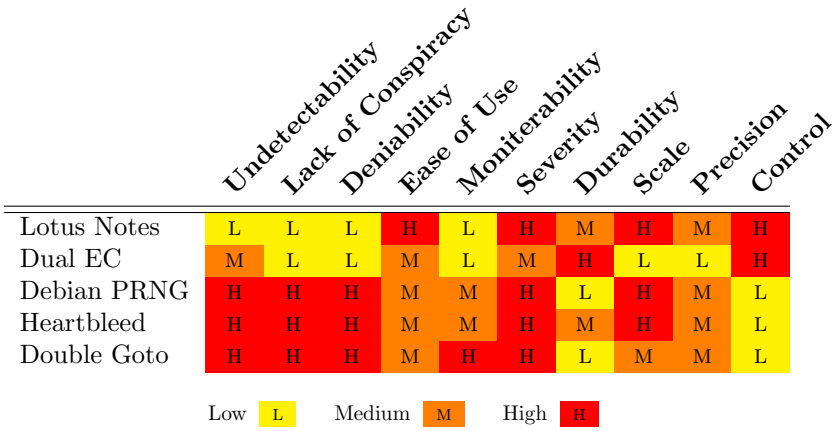

Figure 3: Comparison of historical backdoors along the various dimensions of the taxonomy from Section 4. Low (L), medium (M), and high (H) labels qualitatively gauge the quality of the weakness along the indicated dimension.

a specially crafted message and obtain in return up to 64 KB of data read from the memory of the server process. Since this data may contain host secret keys, passwords for user logins, leftover transcripts from previous connections, and much more, this represents a critical vulnerability. The bug was introduced with OpenSSL v.1.0.1, which was released in January of 2012. As vulnerable versions of OpenSSL were used by somewhere between 24–55% of the Alexa Top 1 Million web servers [30], not only was the bug critical but it also affected a huge fraction of the Internet.

The bug is widely considered to have been introduced unintentionally, putting it in a similar class as the Debian OpenSSL vulnerability. We can easily imagine a saboteur introducing this purposefully, and hence can analyze it according to our taxonomy.

This weakness has low detectability as it went undiscovered for a bit over 2 years, has low conspiracy, and has high deniability. It has high ease-of-use, but also high moniterability as network transcripts containing exploits reveal use of the exploit [30]. The severity is very high. Durability is moderate, as it can be fixed with a one-line change to the source code, but a study done four months after the initial public disclosure showed that 97% of scanned servers had failed to properly remediate the vulnerability [51] The scale was vast, as already discussed above, and precision is low. The control is low as once discovered it can be easily abused by anyone.

Man-in-the-middle enablers. The security of TLS and other protocols relies upon the binding between a public key and the identity of the desired endpoint. This binding is handled by public-key infrastructure (PKI), and security against man-in-the-middle (MITM) attacks arise whenever the PKI is subverted (e.g., via compromise of a CA [52, 63]) or protocol participants fail to check certificates properly. While compromise of a CA is not really a cryptographic weakness, arranging for implementations to fail to check certificates properly can be. There have been a string of high profile vulnerabilities of TLS certificate checking: Apple's double goto bug [24], a bug in OpenSSL [23], and the results of a number of research papers showing certificate checking bugs in a numerous applications [34, 38, 60]. While we have no reason to expect that any of these bugs were maliciously inserted, similar bugs maliciously inserted would be devastating examples of sabotage.

A telling example is the Apple double goto bug, portions of the vulnerability-causing function shown in Figure 4. An extra "`goto fail;`" short-circuits the verification of the server's signature, resulting in a routine that always returns successful. This means that any party can forge signatures and therefore successfully pretend to be the server.

With all the complexity of certificate and signature checking, a saboteur with appropriate access can easily insert an innocuous-looking bug. That means the conspiracy is low (requiring just one insider engineer with access), and the plausible deniability is high. Secrecy can also be high since, assuming the example bugs above are indication, the weakness can unnoticed in implementation for years. These bugs are moderately

```
static OSStatus SSLVerifySignedServerKeyExchange(...)
{
    OSStatus        err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    ...
    return err;
}
```

Figure 4: Vulnerable certificate-checking source code used in some versions of the Apple iOS and OSX operating systems.

easy to use, as they must be performed on-path but are otherwise simple to implement with attack code that consists of (slight modifications to) existing software. Moniterability can be high, assuming visibility into networking traces. Severity is high as it allows complete exposure of sessions, but durability is low as fixes are typically straightforward (in the example above just delete the extraneous line of code). Scale is moderate as it is implementation-specific, and precision is likewise moderate. Control is low.

Discussion. The historical examples looked at above cover a range of approaches and circumstances for surreptitiously weakening cryptographic systems. In the case of Lotus Notes we have an overt inclusion of a weakness into the algorithms and implementations of a proprietary algorithm, with exclusive exploitability by the saboteur. The Debian OpenSSL PRNG and Heartbleed vulnerabilities are examples of how plausibly deniable modifications to a widely used implementation gives attackers broad capabilities. Dual EC illustrates how a standard can be subverted in order to provide, as with Lotus Notes, an exclusive backdoor with low potential for collateral damage. Note that we assume proper management of the trapdoor in these cases, and if it instead leaks or is stolen then control disappears. Finally, MITM attacks show how saboteurs can undermine dependencies required for security.

None of these historical weaknesses achieve maximal benefits to the saboteur/attacker across all dimensions of our taxonomy. Inspection of Figure 3 reveals evidence of revelatory tensions faced by the saboteur. In the examples seen, the cryptographic algorithm must be carefully designed to obtain precision and control (Lotus Notes and Dual EC), while this tends to increase conspiracy and lower deniability.

Finally, it is informative to reflect on how these historical examples fit into the prior academic frameworks discussed in Section 3. Perhaps surprisingly, the Lotus Notes, Dual EC, and MITM examples would all meet the definition of surveillance-resistance under appropriate adaptations of the ASA framework to the relevant primitives: each of these weaknesses exist in the reference algorithm and don't rely on substitution attacks. The Debian PRNG and Heartbleed bugs, on the other hand, can be viewed from the lens of substitution attacks, but don't meet the undetectability requirements mandated in the prior frameworks as an attacker that knows a saboteur's strategy can detect its existence from transcripts. (MITM attacks are typically detectable from transcripts as well.) This suggests that either such bugs should not be considered substitution attacks (as the theory would suggest). A more conservative viewpoint would instead suggest that that high undetectability bar set out by the SETUP (and ASA) frameworks may not really be considered necessary by many saboteurs.

All this is not to state that the SETUP and ASA frameworks are wrong, but rather that we should test the assumptions underlying the frameworks and always be open to exploring alternative models.

Finally, we note that these examples target specific implementations (Debian PRNG, Heartbleed, and MITM) as well as algorithms (Lotus Notes and Dual EC). We will use these observations to drive our analysis of a broader set of (potential) approaches to sabotage.

		Undetectability	Lack of Conspiracy	Deniability	Ease of Use	Severity	Durability	Monitorability	Scale	Precision	Control
Design	Low resistance to cryptanalysis	H	L	M	M	H	H	L	H	L	L
	Backdoor constants	H	L	L	M	M	M	L	H	L	H
	Implementation fragility	H	M	H	L	M	M	L	M	L	L
Implementation	Bad randomness	M	H	H	L	M	L	L	M	M	L
	Leaked key data	H	M	L	H	H	L	L	M	M	L
	Weak key generation	M	H	H	H	H	L	L	M	M	L
	Incorrect error handling	H	H	H	M	M	M	M	M	M	L
	Nonce reuse	L	H	H	M	M	L	L	M	M	L
	Flawed API usage	M	H	H	H	H	L	H	M	M	L

Figure 5: Comparison of approaches for sabotaging crypto systems. We use the same qualitative assessments as in Figure 3 in the various dimensions of our taxonomy.

6 Approaches to Sabotaging Cryptosystems

Educated by historical examples, we now turn to enumerating the various ways in which a saboteur may attempt to weaken a cryptosystem in order to aid one or more attackers. As seen both designs (EC_DRBG, Lotus Notes) and implementations (Debian OpenSSL, Heartbleed, PKI-based MITM) can be targeted by a saboteur. In each category, we attempt to be exhaustive, considering various aspects of cryptosystems that might be susceptible to sabotage. We again assess their potential for being used by saboteurs using the taxonomy of the Section 2. A summary is provided in Figure 5.

6.1 Sabotaging Design

We start by discussing weaknesses that can be built into cryptographic systems at the algorithmic level. The goal of the saboteur here is to force adoption of cryptographic algorithms whose implementation is guaranteed, or highly likely to be, exploitably weak. We focus on weaknesses that are present in "functionally correct" implementations of the system, and can only be fixed by changing the system's parameters or construction. Note that this puts these attacks outside the frameworks previously considered in academic works (e.g., SETUP and ASA attacks, see Section 3).

A saboteur can target designs at two granularities. First, they might introduce a weakness into a public standard. This is generally a high-conspiracy approach that may require substantial social engineering or persuasion. It may not yield weaknesses with high secrecy or deniability, unless the weakness relies on arcane private knowledge (e.g., better-than-publicly-known cryptanalytic methods) or on the inherent complexity of by-committee design processes. That said, this approach can yield the saboteur a valuable asset, meaning a weakness that is difficult to fix and that affects a large number of users. An example from the last section is the Dual EC NIST standard.

Alternatively, a saboteur might target a particular system design. This approach may require less conspiracy and may provide greater deniability. However, a weakness introduced in this way may affect fewer users and may not persist for as long because users may have a choice of which products to use. An example from the last section is Lotus Notes.

Below we investigate several ways a saboteur might weaken the design of a cryptosystem.

Low resilience to cryptanalysis. Designing cryptosystems that are resilient to cryptanalysis is challenging. Therefore, one way to weaken a cryptosystem is to intentionally fail at this task. Part of the difficulty in designing resilient cryptosystems lies in the fact that new cryptanalytic techniques are evolving, and the cutting edge may not be known to the public at design time. This can be exploited by those with advance

knowledge of new cryptanalytic techniques to introduce specific weaknesses that will remain unknown to everyone else. For example, the substitution boxes used in AES were carefully constructed to be resilient to linear and differential cryptanalysis. Rijmen and Preneel [64] discuss the possibility of constructing block ciphers that are susceptible to linear cryptanalysis with high probability, and are computationally infeasible to detect. Recent work by Albertini et al. [2] show how to choose constants for a modified version of SHA-1 enabling a saboteur to help an attacker later find collisions of a certain form. Both of these examples are also examples of choosing weak constants, a strategy discussed below.

With this weakness, exploitability hinges on knowledge of the cryptanalytic attack. While at first glance, this may seem to allow secrecy, history shows that other researchers often reproduce knowledge first developed behind closed doors. Consider the invention of public-key cryptography, first occurring at GCHQ [33] and, only a few years later, in academia [27, 66]. Another example, more directly related to the topic at hand, is differential cryptanalysis as known to IBM and the NSA at least at the time of the design of DES [18] and its later discovery in academia [9].

Another way to ensure weakness to cryptanalysis would be to specify a *security parameter* that is too small to prevent attacks by a party with sufficient computational resources. This approach, however, is not surreptitious as the nature of the weakness is generally obvious to users of the crypto systems. The most familiar such example is the 40-bit key length commonly used in software released for export before 1996, when cryptographic algorithms with larger keys could not be legally exported from the United States without specific clearance from the government [67]. This affected international versions of popular browsers such as Internet Explorer and Netscape Navigator, which could only support SSL with 40-bit keys. As early as 1996, several independent researchers reported breaking 40-bit SSL within 30 hours on standard hardware [39], and within 2 seconds using approximately $100,000 of specialized hardware [67].

- *Secrecy*: The secrecy of these approaches varies depending on the specific mechanism used to insert the backdoor. If advance cryptanalytic knowledge is used to weaken the structure of basic primitives (e.g., weak S-boxes), then it is likely to be impossible to detect the weakness (ignoring the possibility of a leak) until the cryptanalytic knowledge is reinvented publicly (e.g., by academics). On the other hand, weak security parameters are typically non-surreptitious, and researchers have only to demonstrate the feasibility of an attack for a parameter hypothesized to be weak. Deniability is moderate, as the saboteur may be able to argue that the chosen constant or parameter was not known to be weak at the time. The degree of conspiracy needed to install this type of weakness is generally high, as it requires communication between cryptanalysts, designers, and standards committees.
- *Utility*: Depending on the resources of the attacker, this weakness can be useful for a full break of the system. For example, when the DES key length was shortened, it was conceivable that government entities could utilize the weakness to break security. This weakness is not fixable in cases where the parameter length or primitive design is codified in a standard, in which case it is effective until the standard is superseded, but in rare cases may be quickly fixed should users or implementors have control over the parameter. More often, however, deployed crypto systems remain in use for long periods of time even once generally acknowledged as weak. Ease-of-use is moderate since often it requires a reasonably large amount of computational resources to exploit cryptanalytic weaknesses, yet once obtained attacks can be mounted at will. It is unlikely to be moniterable, in the sense that cryptanalytic weaknesses are likely to be exploitable offline with passive attacks.
- *Scope*: If a weak security parameter is part of a widely-used standard, then this weakness will affect a large number of users. However, it offers little precision for the saboteur, and the exclusivity is low whenever other parties have similar resources to the attacker. For example, a key length susceptible to brute-force attack by the NSA might also be susceptible to attack by other governments with wealthy intelligence agencies. When the weakness involves arcane cryptanalytic weaknesses, thus is impossible to control.

Backdoor constants. Much of the time, especially in asymmetric cryptography, the security of the system relies on selecting mathematical constants that do not exhibit certain properties that would weaken the cryptosystem. For example, the constants used to define elliptic curves can in some cases be chosen in such a way as to trivialize cryptanalysis for the party who chose the curve. A bad constant can be codified in a public standard, as is possibly the case with NIST's Dual EC pseudorandom number generator, or inserted

by a programmer whenever the standard does not dictate a specific constant. The majority of protocols that rely on trapdoor common-reference strings [13] are, by design, trivial to sabotage. The weakened S-box constants [64] and SHA-1 round constants [2] mentioned above are other examples.

- *Secrecy*: If the role of the constant in the security of the system is not well-understood, then this weakness can be both secret and deniable. If the constants are one for which a trapdoor exists as with Dual EC, then the plausibility deniability is low. Constants are often called "nothing-up-the-sleeve" if they are selected in some demonstrable manner, such as picking numbers from the numerical representation of well known mathematical constants such as π. Constants that are randomly selected can therefore be a tell-tale sign of foul play, and lower secrecy.
- *Utility*: Depending on the technical structure of the bad constant and its relationship to higher-level protocols, this weakness can result in a total break of the system. For example, the Dual EC weakness requires knowledge of a full block of PRNG output (30 bytes in most cases), which is two bytes more than most TLS implementations provide; this makes exploitation of the full protocol more expensive [16]. As with poor security parameter values, the duration and fixability of this weakness depend on whether the constant is codified in a public standard.
- *Scope*: Anyone who uses an implementation containing the bad constant will be affected, as with the security parameter weakness. The collateral risk depends on specific properties of the constant. For example, in the hypothetical DUAL_EC_DRBG weakness, the attacker would need to know another constant to utilize the weakness, so in this case collateral risk is conceivably low.

Implementation fragility. One way to sabotage a cryptographic design is to ensure that implementing it in a secure way is exceedingly difficult and error-prone. This could be accomplished via selection of algorithms that are difficult to implement without side channels (e.g., AES [57], GCM [45]) or other critical deficiencies. For example, if the "nonce" used in a DSA signature is predictable, or if the same nonce is used in two signatures signed from the same key, then it is possible for an attacker to recover the persistent signing key from just one or two signatures [59]. To prevent this, the implementor must be careful to treat the DSA nonce as a high-entropy session key rather than a "nonce" as the term is traditionally used. The complexity of the PKI ecosystem and the difficulty of correctly implementing certificate checks often lead to egregious mistakes [24, 34, 38].

This also may arise at the protocol standardization level. Many important standards such as IPsec, TLS and others are lamented as being bloated, overly complex, and poorly designed (c.f., [37]), with responsibility often laid at the public committee-oriented design approach. Complexity may simply be a fundamental outcome of design-by-committee, but a saboteur might also attempt to steer the public process towards a fragile design. In addition, saboteurs might also simply seek protocol designs that are hard to use at all (let alone use securely!). Usability is, of course, a widely-acknowledged challenge for cryptography (c.f., [77]) and reducing usability would seem easy.

- *Secrecy*: The undetectability of such a weakness is high, as even for benign settings the community has a poor understanding of implementation complexity or fragility. Conspiracy is moderate as maliciously shifting a cryptosystem towards implementation fragility may require being involved in standardization groups and the like, but we do not mark it as high since hard-to-implement cryptosystems arise even without a saboteur. The plausible deniability is very high, as the actual weakness will likely be blamed on the committee process leading to overly complex design, or blame may rest with the implementor.
- *Utility*: This type of weakness tends to have lower usability, as it relies on an implementor failing in a particular way or on attackers discovering vulnerabilities in each implementation. Even when the implementor fails as intended, attacks such as side channels can be fairly difficult to mount in many settings.
- *Scope*: As with any design that gains wide approval, the scope here can be large, though some implementors may not fall into the traps laid by the saboteur so weakness is not guaranteed everywhere. Precision is low since the saboteur cannot easily control who becomes vulnerable, and collateral risk is often very high, as anyone can take advantage of the sorts of weaknesses introduced by the resulting insecure implementations.

6.2 Flawed Implementations of Secure Cryptosystems

In this subsection, we discuss weaknesses that arise from flawed implementations of cryptosystems that are otherwise secure. To distinguish from the previous section, here we refer to strategies that target inserting a specifically designed weakness into implementations of a cryptosystem. The design of the system may indeed be sound. Substitution attacks fall into this category, as they seek to replace a secure implementation with a backdoored one. A saboteur might target both hardware and software; our discussion will be high-level and agnostic to this distinction.

Bad randomness. Random numbers are critical for cryptography: key generation, random authentication challenges, initialization vectors, nonces, key-agreement schemes, prime number generation, and many other processes rely on the availability of high-entropy randomness. At the same time, one of the hardest parts of cryptography is random number generation — it is not difficult to write a bad random number generator, and most of the time it is not obviously bad. Bad randomness almost never prevents proper functioning of a crypto system, and statistical testing for randomness quality (e.g., [55]) may fail to detect cryptographically weak randomness. Weak random number generators have been a perennial problem in cryptographic engineering, even in benign settings, with the result often being easily-crackable keys (c.f., [1, 31, 39, 42, 65]).

- *Secrecy*: In general, introducing a flaw into the random number generator is "safer" than directly modifying other parts of the cryptosystem that can be tested against alternate implementations. Depending on the subtlety of the bug causing bad randomness, the secrecy and plausible deniability of this weakness can be high. For example, the Debian OpenSSL bug [1] went unnoticed for two years, and could have been easily denied by an intentional saboteur.
- *Utility*: The utility of this weakness depends on the way in which it is used by other cryptographic routines. If it is not properly mixed with other sources of entropy and used to generate key data or nonces that repeat with high probability, then it can result in a total break of security. On the other hand, if the bad source is never used in isolation, or the total reduction in entropy is small, then this weakness will have marginal utility.
- *Scope*: This weakness offers potentially high precision, as one can use it with individual implementations, but offers no control over exploitability of the weakness when it ruins entropy of random values for all users.

Leaked key data. It is possible to modify existing cryptosystems so that key data is leaked in the contents of ciphertexts. For example, an implementation could "accidentally" reuse a key from one encryption instance as a nonce in another. Accomplishing key leakage in a robust and secure (from those other than the saboteur) manner is the goal of *subliminal channels* and *kleptography*. In a series of early results, Simmons showed that subliminal channels could be inserted into the Schnorr, ElGamal, and DSA signature algorithms to leak signing key data [71, 72]. This contradicted the widely-held belief at the time that signature algorithms could not be used for secret communication. In a later set of results, Young and Yung showed a number of *kleptographic* attacks that cause cryptosystems to leak key data under the saboteur's public key. Generally, these attacks work by subverting the randomness used to generate private primes in asymmetric schemes, but they also showed how symmetric schemes could be designed to leak key data with the same properties [83]. They showed how to introduce these attacks into RSA, Diffie-Hellman exchange, DSA, and ElGamal [80–82].

- *Secrecy*: Key leakage attacks can be resilient to black-box defenders who have access only to the input/output behavior of the cryptosystem. They require a substantial amount of engineering, and so could be easy to detect in an open-source application, and are not plausibly-deniable. Outsiders may not be able to detect their use, however, given that one can design them to be indistinguishable from a normally functioning crypto system.
- *Utility*: A successful key leakage completely subverts the security of the cryptosystem. As described by Simmons [73], Young, and Yung [80], subliminal channels and kleptographic attacks are easy for an adversary to use given access to ciphertext data and whatever key material was used to secure the subliminal/kleptographic channel. Fixing this weakness can be accomplished by switching to an unaffected implementation or cryptosystem.

- *Scope*: If the weakness is inserted into a particular implementation, then only the users of that implementation will be affected. The control can be high with subliminal channels and kleptographic attacks, as they are designed to be secure against those other than the saboteur.

Leak sensitive data. Even if an implementation has not been replaced with one that is designed to leak keys, it can be that other weaknesses nevertheless give rise to attackers' ability to glean useful information such as (partial) key data, plaintext information, or other important values. Unlike with kleptography-style attacks, here we focus more on implementation factors that make an implementation susceptible to sensitive data leaking. There are several subtle ways in which data can be leaked. Side-channel attacks are one possible source of inspiration for saboteurs. Researchers have repeatedly abused implementations of arithmetic algorithms in asymmetric systems, such as square-and-multiply, that exhibit side channels from which one can extract information about private key data [26]. In 2012, a flaw in Google Keyczar was discovered [54] where a naive method for comparing HMAC results yielded a harmful side channel: based on how long a failed comparison takes to compute, an adversary is given sufficient information to forge signatures on SHA-1 HMAC messages.

Another route for saboteurs is to have implementations improperly handle plaintext and key data. For example, some implementations fail to ensure that plaintext or key data is destroyed after encryption. Other systems use temporary files to protect against data loss during a system crash, or virtual memory to increase the available memory; these features can inadvertently leave plaintext lying around on the hard drive [21]. In extreme cases, the operating system may leave keys on persistent storage or attackers may be able to extract sensitive data 2 only in memory [17, 40]. An excellent example of the latter is the recent Heartbleed vulnerability, in which a buffer overread vulnerability allowed remote attackers to extract sensitive content from the process memory of OpenSSL implementations. In all these cases, it does not matter how good the cryptographic design is; security is compromised by the details of the implementation.

- *Secrecy*: The wide range of possible ways to insert this weakness lead to varying amounts of secrecy. Subtle timing and power side-channels are oftentimes introduced unwittingly by experts, and thus may conceivably be difficult even for other experts to identify. Similarly, the deniability of this weakness is tied to the subtlety of the leak.
- *Utility*: Timing and power side channels, as well as more esoteric techniques like cold-boot attacks [40], are often difficult for all but very powerful attackers to exploit. These channels also may not yield perfect fidelity, instead leaking a subset of the pertinent bits. Other leaks, such as those involving temporary files and virtual memory, are easier to use, and are more likely to leak all of the available sensitive information.
- *Scope*: This type of weakness will affect all users of the flawed implementation. Furthermore, assuming the leaked data is not first encrypted with the saboteur's public key as in subliminal channels and kleptographic attacks, anyone who is able to intercept the channel will be able to exploit the weakness, so collateral risk is high.

Incorrect error handling. In some cases, the behavior of the implementation on encountering various types of errors can leak enough information to break security. One of the most well-known instances of this weakness occurred in ASP.NET, and was discovered in 2010. When ASP.NET used CBC mode encryption, it returned a status indicator that revealed whether or not a ciphertext decrypted to a message with valid padding [29]. An adversary who is able to submit many ciphertexts can use this information to reconstruct information about the intermediate state of the decryption operation, and subsequently learn the contents of a given ciphertext.

- *Secrecy*: The flaws enabling this weakness often seem completely innocent: they simply return information that may be useful in diagnosing a problem with the system. This type of weakness can be very difficult to detect unless an implementor is disciplined enough to avoid returning any status information to un-authenticated parties, because it is usually not clear when an error message leaks a dangerous amount of information. Additionally, because error messages that may have been useful for debugging can easily remain in code that is eventually released, this weakness offers plausible deniability.

- *Utility*: In most cases, the attacker will need to interact with the system in an anomalous fashion over several rounds of communication. This makes it possible for a defender to detect and throttle an active attempt to exploit, possibly even without direct knowledge of the flaw. Additionally, the attacker might only gain incremental knowledge with each additional round of interaction.
- *Scope*: All users of the flawed implementation will be affected, and the saboteur does not have control over who exploits the weakness once it is publicly known.

Reuse Nonces and Initialization Vectors. A common mistake when implementing cryptographic protocols is to reuse nonce values across different sessions or messages. Although nonces generally do not need to be secret or unpredictable, if an implementation allows nonces to be reused, then in some cases the implementation becomes vulnerable to replay or key/plaintext disclosure attacks. A similar concern arises for certain modes of block cipher operation (e.g., CBC and CFB) with initialization vectors — reusing values can leak some information about the first block of plaintext, or whether ciphertexts share a prefix. These weaknesses are particularly insidious, as they may result from subtle flaws such as generating nonce and IV data from insufficient entropy, or mistakenly rolling over a counter.

- *Secrecy*: Being that misuse of these values is a historically common problem, conspiracy is low and plausible deniability high, It has low undetectability, however, since reused nonces can most often be detected on the wire.
- *Utility*: This weakness can be used to passively leak partial information about a set of plaintexts (when IV values are reused), or actively to perpetrate a replay attack. While the former is easier to use, the latter requires more resources and expertise but is potentially more severe depending on the application.
- *Scope*: This weakness typically must be embedded in specific implementations, offering moderate scope and precision. Also there is basically no control, once detected, anyone will be able to exploit reused nonces or IVs.

Flawed API Usage. Conventional wisdom advises against re-implementing cryptographic functionality when well-known and widely-vetted libraries exist for accomplishing the same task. Open-source crypto libraries are available for most of the functionality commonly needed by developers seeking to secure their code, and often have the benefit of being written and examined by people with specialized knowledge and experience in the field. However, the complexity of many protocols and modes of operation leaves significant room for error even when a developer selects a good library. Developers who make incorrect use of flags, select the wrong mode of operation or primitive, or fail to properly account for status indicators can negate the security guarantees purportedly granted by the library. For example, in 2013 Egele *et al.* found that the predominant Android security API defaulted to ECB mode for AES encryption [32]. Although the library might implement this mode correctly, users who are unaware that encrypting sufficiently large blocks of data in this way can leak a large amount of information might unintentionally implement weak security. In 2012, Georgiev *et al.* [38] documented several instances of commercial software that used SSL libraries incorrectly, effectively ignoring certificate validation checks and thus leaving their customers open to easy man-in-the-middle attacks.

- *Secrecy*: The findings of Georgiev *et al.* [38] suggest that these weaknesses might be quite prevalent, so it is likely that they commonly go unnoticed in many implementations. The relative complexity of most cryptographic APIs provides plausible deniability, as the risk of innocent programmer error seems high.
- *Utility*: The utility of this weakness depends on which API is misused, as well as the way the application ultimately uses the outputs of the library. For an application that uses ECB-mode encryption to transmit large images over insecure channels, it may be easy for a passive attacker to infer a significant amount of information. For applications that misuse certificate validation checks, the attacker needs to perpetrate an active man-in-the-middle attack.
- *Scope*: The precision of such weaknesses can be relatively high, should it be added to a single user's implementation. The scale also varies according to means of insertion. The weakness, however, has low control as anyone aware of the flaw will typically be able to exploit it.

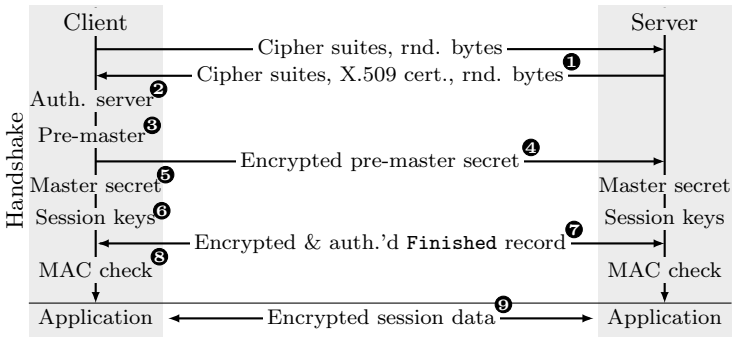

Figure 6: Overview of TLS protocol using RSA key transport.

7 Case Studies: TLS and BitLocker

To illustrate the enumeration of potential cryptographic weaknesses presented above, we investigate in more detail two hypothetical targets for sabotage: TLS and BitLocker. These systems provide different architectures and security goals that together are representative of many of the kinds of software-based cryptographic systems found in practice.

7.1 TLS

TLS is the predominant protocol for providing link security on the Internet. It supports many modes of operation, including authentication for both client and server, "resumed" sessions, and an array of cipher suite options for greater flexibility. The basic protocol, which only authenticates the server, is outlined in Figure 6.

TLS proceeds in two phases, referred to as the *handshake* and *record* sub-protocols. When a client initiates a connection, it initiates the handshake by sending its TLS version number, a list of supported cipher suites, and a set of random bytes. The server responds by selecting a cipher suite from the client's list, sending an X.509 certificate containing its public key, and its own random bytes. After the client verifies the server's certificate, it generates a *pre-master secret*, and uses the server's public key (given in its X.509 certificate) to encrypt before handing it back to the server. Note that if the client and server choose, they can also use Diffie-Hellman to generate the pre-master secret, rather than having the client explicitly send it; in this discussion we will assume that the Diffie-Hellman exchange is not used. At this point, both parties perform a sequence of identical computations to generate a *master secret* from the pre- master, and subsequently a set of *session keys* to be used by the target application for symmetric encryption in later communications. Before the handshake completes, both parties construct a `Finished` message by computing a hash and MAC of the previous handshake transcript. Both parties verify the other's `Finished` message, and if successful, pass control to the application. The application uses TLS's record protocol, which employs the symmetric session keys derived from the master secret, as well as authentication checks like those used for the `Finished` messages, to provide secure communications on the link.

There are several goals an attacker might have regarding TLS connections, including decryption of ciphertexts, impersonation of either party, and replaying sessions, among others. In this discussion, we will focus on the first threat. A saboteur can enable the attacker by compromising several components of the TLS protocol stack:

- *Secure randomness:* The security of the master secret (components 3 and 5 in Figure 6), and thus all of the session keys (component 6), depends on the quality and secrecy of the entropy used to generate them.
- *Underlying primitives:* Because most TLS implementations must inter-operate with others, the saboteur may not be able to assume that a "backdoored" implementation of basic cryptographic primitives (e.g., encryption, key exchange, hash functions) would go unnoticed. However, opportunities remain

with certificate and HMAC verification routines (components 2, 8, and 9), which do not require interoperability.

- *Public-key infrastructure:* TLS uses X.509 certificates, so trusted certificate authorities are necessary for authentication (component 2), and to prevent man-in-the-middle attacks on secrecy (components 4 and 9).
- *Error oracles:* TLS is a complex protocol with many possible failure modes. If the implementation reports excessive information about certain types of errors, such as CBC padding failures [75], then the attacker may be able to infer enough information about the private state to mount an attack.

We discuss each of these opportunities in more detail below.

Secure randomness: Secure generation of random bytes plays a large role in initiating TLS sessions. As previously discussed, the client is responsible for generating the *pre-master secret*, which is the only secret value used to deterministically generate the master secret, and subsequently the session keys used to encrypt all application data. If the attacker can predict the pre-master secret, then all subsequent secrecy is totally compromised. Therefore, replacing the PRNG used to generate the pre-master secret with one that outputs predictable bytes is an excellent approach for the saboteur.

One way to weaken the PRNG is to limit its source entropy. Because the rest of the PRNG is deterministic, this will restrict the values taken by the pre-master secret to a small domain, so that the attacker can efficiently guess the correct value for any session. It has low detectability, because the output of a PRNG weakened in this way will look "sensible" (i.e., random). Furthermore, it may also have low conspiracy and high plausible deniability. The primary drawback to this approach is its scope and fixability — it must be introduced at the level of an individual implementation, and is trivial to fix once discovered.

This is exactly what happened with the Debian OpenSSL PRNG bug, which resulted in a source entropy pool containing at most 2^{15} bits, and thus 2^{15} possible initial states for the PRNG. If the attacker knows the sequence of calls to the flawed OpenSSL PRNG, then he can guess the pre-master secret in as many attempts. This is a serious concern for clients that initiate a small number of SSL connections, e.g., `wget` and `curl`, which often initiate a single connection before terminating. However, as no major browsers use OpenSSL, the relevance of the bug to this particular aspect of TLS security was limited.

Another way to weaken the PRNG is to ensure that its *internal state* can be deduced after observing some number of bytes produced as output. This approach may be effective with TLS, because raw output from the PRNG is available to outside parties in several ways:

- Both the client and server transmit 28 bytes of cleartext random data in their respective `Hello` messages. Some implementations, such as the NSS library and certain branches of OpenSSL, have begun transmitting 32 bytes in the `Hello` messages by replacing the four bytes commonly reserved for the current system time, as the system time can be used to de-anonymize a party that sends it in the clear.
- The client transmits 48 bytes of random data to the server in the form of the pre-master secret, so if it is possible to coerce the client into performing a TLS handshake with a malicious server, then such a weakness could be used to compromise future sessions from that client.
- The client transmits an indeterminate number of random bytes to the server in the form of padding for the encrypted pre-master secret. As with the pre-master secret, this is an opportunity for an adversary able to coerce a handshake from the client.

Making the internal state predictable in this way does not rely on introducing a flaw into a single implementation of the PRNG, so it has the potential for larger scope as well as a higher conspiracy cost, assuming it is introduced as a standard or public specification (as in the case of Dual_EC_DRBG). It may also be more difficult to detect and fix, as doing so would require understanding a deeper flaw in the underlying logic of the PRNG and finding a suitable and well-supported replacement PRNG.

This type of weakness is embodied in the design of Dual_EC_DRBG—after the attacker observes a single block of contiguous output from the generator, he can recover its internal state in approximately 2^{15} attempts [69] (the exact complexity depends on the particular block used in the recovery, as well as

the constants used to define the generator's elliptic curve). However, complications for the attacker have been shown to arise in TLS implementations [16]. The most typical block size used in implementations of Dual_EC_DRBG is 30 bytes, while the random field in the TLS `Hello` message is only 28 bytes (in most TLS implementations); this raises the complexity of the attack to 2^{31}, which is still feasible, but certainly degrades the scalability of the attack. Thus, the weakness is more useful in settings where the attacker can coerce the client into performing a handshake, obtaining at least 48 bytes of output from the PRNG.

Underlying Primitives: Compromising the cryptographic primitives used in TLS is more challenging, as the protocol uses a pre-defined list of acceptable ciphersuites that are known to the general public. The compromised primitive must be input-output compatible with non-compromised versions of these primitives, so flawed implementations are unlikely to work with deterministic primitives. Two exceptions to this reasoning lie in the certificate verification and HMAC verification routine used to check the `Finished` messages as well as record-layer messages. The certificate verification routine could be compromised to return True all the time, or when certificates with certain properties are presented, thus allowing man-in-the-middle attacks. Similarly, the HMAC verification could be subverted to return True when it should not, thus allowing an adversary to modify the contents of either the handshake messages or encrypted application messages.

Several instances of the first type of weakness, wherein certificate verification succeeds when it should not, were discovered in 2012 by Georgiev *et al.* [38]. While the verification primitives themselves were not directly compromised, several higher-level API routines that use them were found to employ confusing interfaces that obscured the functionality of the verification primitives, effectively contradicting their intended purpose. Additionally, Georgiev *et al.* observed several high-profile implementations that did not invoke the primitives correctly, achieving the same effect as an intentionally-weakened implementation—complete vulnerability to man-in-the-middle attacks from arbitrary parties.

Modification of handshake messages could allow a *cipher-suite rollback* attack, wherein the adversary modifies the list of available ciphers sent from one party to include only weak candidates. Notice that the list of cipher suits sent by both parties in Figure 6 are unencrypted, so modifying their contents is trivial for a man-in-the-middle attacker as long as the verification performed after the `Finished` message complies. If the attacker possesses cryptanalytic abilities on one of the ciphersuites supported by both parties, then the severity of this attack could be quite high.

Both of these weaknesses may be difficult to detect without manual source review, particularly if they only falsely return True when certificates or HMAC values satisfy properties that are unlikely to appear in a routine black-box analysis of the implementation. Thus, they may enjoy low detectability. Depending on the context, a weakness that behaves in this way might suffer from low plausible-deniability; however, as Georgiev *et al.* demonstrated, if the weaknesses are introduced to mimic a simple misunderstanding of API semantics, this is not true. Both weaknesses also have limited scope, as they rely on a code-level deviation from a public standard. The collateral risk of this approach may also be high, depending on how often the verification procedure incorrectly returns True.

7.2 BitLocker (Disk Encryption)

BitLocker is a full-disk encryption utility included in many versions of Microsoft Windows since the Vista release. By default, it encrypts a sector by composing a proprietary diffuser named "Elephant" [36] with a pass of AES in CBC mode. The purpose of the diffuser is to provide enhanced authentication for the ciphertexts; due to engineering constraints, other types of authentication such as MACs and nonces cannot be efficiently used in this setting, and BitLocker's designers settled on this form of "poor man's" authentication [36]. BitLocker supports many modes of authentication, including a "transparent" mode requiring no user interaction for normal operation. This is accomplished with the use of a trusted platform manager (TPM), and is depicted in Figure 7. During the boot sequence, the operating system loader makes a request for the TPM key, which is only released if critical early-boot components, stored in the TPM's *platform configuration registers* (PCR), pass an integrity check. The TPM key is used to decrypt the *master key*, which is in turn used to decrypt the *volume key* under which the contents of the hard drive are encrypted. BitLocker uses two keys to streamline the process of re-keying the system when upstream keys are compromised [36]. Once the volume key is decrypted, it is given to BitLocker's *filter driver*, which automatically encrypts and decrypts hard drive contents as they are needed by the filesystem driver. A system running

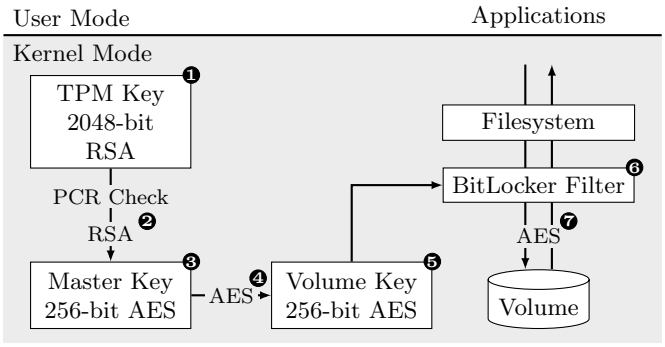

Figure 7: Architecture of BitLocker with TPM authentication.

1. Pick random primes p, q, set $n \leftarrow pq$.
2. Repeat until $\gcd(e, \phi(n)) = 1$:
   (a) Pick a random odd δ such that $\gcd(\delta, \phi(n)) = 1$ and $|\delta| < k/4$.
   (b) Set $\epsilon \leftarrow \delta^{-1} \mod \phi(n)$, $e \leftarrow \pi(\epsilon)$
3. Set $d \leftarrow e^{-1} \mod \phi(n)$
4. Return (p, q, d, e)

Figure 8: Crépeau and Slakmon's [20] RSA key generator with backdoor based on random permutation π.

BitLocker requires an additional partition that remains unencrypted, from which the system boots and performs additional integrity checks.

The goal of an attacker is to decrypt the contents of the disk. To aid in this task, several of BitLocker's cryptographic components could be surreptitiously compromised by a saboteur. We group them into two categories:

- *Underlying primitives:* By default, BitLocker uses both RSA and AES encryption to protect key data and hard drive contents, as well as the proprietary Elephant diffuser (components 2, 4, and 7 in Figure 7).
- *Persistent key data:* Components 1, 3, and 5 in Figure 7 represent points at which BitLocker's designers chose to store long-term keys for use across multiple sessions. If any of these keys is leaked, then the security of BitLocker is compromised.

Underlying Primitives: At first glance, it seems as though a system like BitLocker is a prime candidate for a weakened encryption/decryption primitive that does not correctly implement AES. All data is encrypted by BitLocker, and BitLocker is the only system responsible for decrypting it. Furthermore, the system is closed-source, and resides at a low level in the software stack, so reverse-engineering might be difficult. However, the BitLocker developers have published information sufficient for others to re-implement BitLocker's decryption routines, which several forensics and password recovery tools have readily done [50], so this is an unlikely route for a saboteur.

A more promising primitive is the key generator used in the TPM's RSA module. Several researchers [4,20, 80] have demonstrated the possibility of generating keys in such a way that the designer of the key generator can easily factor the modulus (with his own "backdoor" key), but others cannot. One such scheme was given by Crépeau and Slakmon [20] in 2003, and is based on Wiener's attack on small private exponents [78], shown in Figure 8. It makes use of a random permutation π, and works as follows for modulus of length k.

An attacker can recover p and q by first applying the inverse permutation $\pi^{-1}(e)$ to learn ϵ, and then using Wiener's attack to compute δ from ϵ and n. The private prime factors p and q are then recovered by a

straightforward factorization of n using ϵ, δ. Crépeau and Slakmon argue that as long as π can be computed quickly, the difference in runtime between this algorithm and the standard RSA key generation algorithm is quite small.

Once the attacker is able to learn p and q, recovering the master key is trivial, so the severity of this approach is high. Assuming the runtime of the weakened key generation algorithm cannot be distinguished from that of the standard algorithm, discovering this backdoor requires reverse-engineering the TPM hardware. Although this has been done on at least one TPM [74], it requires expensive equipment and sophisticated techniques. Thus, this approach has low detectability and low fixability. However, if reverse engineering is successful, the non-standard procedure is "smoking gun" evidence of a backdoor, so it also has low plausible-deniability. As long as the permutation π is hard to guess, the approach has low collateral damage, and the scale/precision of the backdoor can vary according to the saboteur's resources and requirements.

Persistent Key Data: When a machine running BitLocker is turned off, the master and volume keys are both encrypted either directly or indirectly via an RSA key that resides in the TPM. However, when the machine is running, the volume keys are present as cleartext in kernel memory [41]; this is necessary for efficient encryption/decryption of sector data. One approach for compromising BitLocker's security is to read these keys from kernel memory, and write them to the unencrypted boot partition. If the saboteur wishes to prevent collateral damage, the keys should first encrypted with the attacker's public key. This approach can be very difficult to detect, as the code responsible for implementing it can reside anywhere in the operating system, or even within a third-party application, and it can expunge itself from the system once the keys have been written to the boot partition. However, once discovered, this approach has low plausible deniability, as no part of BitLocker should behave in this way.

8 Towards Sabotage-Resistant Cryptography

The prior sections highlight the many vectors by which cryptographic systems can be weakened maliciously. Most academic work thus far has focused on showing feasibility of backdoors, and relatively little attention have been given to building backdoor-free cryptography. More is needed, and we below highlight several potential concrete next steps.

(1) Frameworks for understanding whitebox design weaknesses. Thus far the only formal frameworks aimed at understanding backdoors has been in substitution attack settings [6, 80]: the saboteur arranges for a subverted algorithm to replace a correct one. These attacks are assumed to be black-box, meaning defenders only have API access to the algorithm (whether correct or subverted). Yet, many of the confirmed examples of sabotage are directly built into public designs (e.g., EC_DRBG). In these settings substitution attacks are inapplicable. We might refer to these as 'whitebox design weaknesses' since they withstand knowledge and scrutiny of the cryptosystem design.

A starting point for future research would be to attempt to develop formal models analogous to the SETUP and ASA models, but for this white-box setting. We are not yet sure what such a framework would look like, but we know that it must not assume that the subverted version of a system is not functionally identical to a non-backdoored version.

(2) Robustness in provable-security design. Much of modern cryptographic design has focused on security definitions and proofs that constructions achieve them. This is a powerful approach, but does not always account for robustness when security definitions are narrow. The goal of semantic security for encryption provides, along one dimension, broad guarantees that nothing is leaked about plaintexts. But schemes proven secure relative to semantic security may suffer from padding oracle attacks [12, 75], various side-channels [48, 49, 62], randomness failures, and implementation faults [14]. In each case, attackers sidestep the strengths of achieving semantic security by taking advantage of something outside the model. That a model fails to account for all attack vectors is not really surprising, since it must be simple enough to admit formal reasoning.

Another issue is that provable security can in some cases encourage designs that are amenable to sabotage. Take for example the long line of work on building cryptographic protocols from a setup assumption such as

common reference strings (CRS) [22]. This is seen as a way to increase security and simplicity of protocols, by avoiding more complex setup assumptions or the random oracle model. To support proofs, the CRS must often be generated in such a way that whomever chooses it can retain a trapdoor secret (analogously to the EC_DRBG parameters). The resulting cryptographic protocol is ready-made for sabotage.

To deal with these issues, we advocate robustness as a general principle of provable-security cryptographic design. By this we mean that a scheme proposed for deployment should be (formally) analyzed relative to many security models, including orthogonal ones. By focusing on multiple models, one can keep each relatively simple, and yet provide more coverage of threats. Examples in this spirit from existing work include [5, 7]. In addition, theoreticians should evaluate each model with regards to its ability to prevent sabotage. Does it rely on subvertible global parameters? How many implicit assumptions are built in? What happens to security if one violates these assumptions? A scheme that relies on getting too many things 'right' is worse than one that has fewer dependencies.

(3) New cryptographic standardization processes. We need to improve our understanding of, and methodologies for, designing cryptographic standards. For widely used higher-level cryptographic protocols (e.g., TLS, WiMax, IPsec), we end up using standards that are designed by public committees such as those formed by the IETF. While we do not want to denigrate the hard work and expertise of those involved, it's clear that there are limitations to this approach and the results can sometimes be disappointing. Both TLS and IPsec are examples of painfully complex and difficult-to-implement-correctly standards [37]. There are many other examples of broken standards, and likely many others for which analysis would reveal problems.

We advocate research and experimentation on new design approaches for cryptographic standards. One approach that has seemingly worked well for low-level primitives is that of public design competitions such as those conducted by NIST. This forces public review, allows a number of small design teams unfettered during design by large committees, and ultimately appears to have yielded strong primitives such as AES and SHA-3. Whether this can work for the more nuanced setting of higher-level cryptographic systems is an interesting open question. A key challenge we foresee will be specifying sound requirements. Even with committee designs, we might begin including in existing standardization processes an explicit review step for resilience to sabotage (perhaps helped by using the outcomes of the hoped-for models discussed above).

(4) Software engineering for cryptography. Implementing even good standards securely remains a monumental challenge. While some notable work has been done on implementation security, such as verified cryptographic implementations [3, 8, 10, 25] and other tools for finding (benign) bugs or well-known problems [15, 31, 43], there seems too little relative to the importance of the topic.

We advocate building such a community. First steps might include workshops bringing together those who have done work in the area and other interested parties. It may also be worthwhile to engage funding agencies such as NSF, DARPA, the ERC, and others about introducing programs aimed at encouraging research in this area. We might also initiate educational programs like the underhanded C contest [19] that would task participants to backdoor a reference piece of cryptographic code.

In the nearer term, we also relay the following pragmatic suggestions from [68]. First, vendors should make their encryption code public, including the specifications of protocols. This will allow others to examine the code for vulnerabilities. While it is true we won't know for sure if the code we're seeing is the code that's actually used in the application, it nevertheless forces saboteurs to surreptitiously substitute implementations. This raises the bar and forces the owner of the system to outright lie about what implementation is being used. All this increases the number of people required for the sabotage conspiracy to work. The community should target creating independent compatible versions of cryptographic systems. This helps check that any individual one is operating properly.

(5) Usability and deployability of cryptography. It is somewhat obvious, but still worthwhile, to say that we need better metrics and design principles for usability of cryptographic tools. This encompasses not only end-user issues, such as those covered in seminal studies on email encryption [77] but also deployability for administrators and others. Setting up IPsec and TLS, for example, can be a hurdle too high for many. Like with software engineering for cryptography, we need a research community that focuses on ensuring software can be deployed and used. Our suggestions for getting there are similar to those above.

References

[1] Debian security advisory: OpenSSL predictable random number generator. Available at `http://www.debian.org/security/2008/dsa-1571`, May 2008.

[2] A. Albertini, J. P. Aumasson, M. Eichlseder, F. Mendel, and M. Schläffer. Malicious Hashing: Eves Variant of SHA-1. In *Selected Areas in Cryptography–SAC*, 2014.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: Efficient provably secure machine code from high-level implementations. In *ACM Conference on Computer and Communications Security–CCS*, pages 1217–1230. ACM, 2013.

[4] R.J. Anderson. A practical RSA trapdoor. *Electronic Letters*, 29, May 1993.

[5] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In *Advances in Cryptology–ASIACRYPT*, pages 232–249. Springer, 2009.

[6] Mihir Bellare, Kenneth G Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In *Advances in Cryptology–CRYPTO*, pages 1–19. Springer, 2014.

[7] Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In *Advances in Cryptology–ASIACRYPT*, pages 299–314. Springer, 2006.

[8] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for tls. In *ACM Conference on Computer and Communications Security–CCS*, pages 459–468. ACM, 2008.

[9] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72, 1991.

[10] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *IEEE Computer Security Foundations Workshop*, pages 0082–0082. IEEE Computer Society, 2001.

[11] Matt Blaze and Joan Feigenbaum. Master-key cryptosystems. DIMACS Technical Report TR-96-02, 1996.

[12] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology–CRYPTO*, pages 1–12. Springer, 1998.

[13] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *ACM Symposium on Theory of Computing–STOC*, pages 103–112. ACM, 1988.

[14] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology–EUROCRYPT*, pages 37–51. Springer, 1997.

[15] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy*. IEEE, 2014.

[16] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Schacham. On the practical exploitability of Dual EC in TLS implementations. In *USENIX Security Symposium*, 2014.

[17] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004.

[18] D. Coppersmith. The data encryption standard (DES) and its strength against attacks. *IBM J. Res. Dev.*, 38(3):243–250, May 1994.

[19] Scott Craver. The underhanded C contest. `http://underhanded.xcott.com/`.

[20] Claude Crèpeau and Alain Slakmon. Simple backdoors for RSA key generation. In *Topics in Cryptology*, volume 2612. 2003.

[21] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating encrypted and deniable file systems: Truecrypt V5.1a and the case of the tattling OS and applications. In *Proceedings of the 3rd Conference on Hot Topics in Security*, 2008.

[22] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In *Advances in Cryptology–EUROCRYPT*, pages 418–430. Springer, 2000.

[23] NIST National Vulnerabilities Database. Vulnerability summary for CVE-2014-0224, 2014. Available at `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0224`.

[24] NIST National Vulnerabilities Database. Vulnerability summary for CVE-2014-1266, 2014. Available at `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266`.

[25] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11. In *IEEE Computer Security Foundations Symposium–CSF*, pages 331–344. IEEE, 2008.

[26] Jean-Franois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestr, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *Smart Card Research and Applications*, volume 1820, pages 167–182. Springer Berlin Heidelberg, 2000.

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

[28] Whitfield Diffie and Martin E. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, 1977.

[29] Thai Duong and J. Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. May 2011.

[30] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of Heartbleed. In *ACM Internet Measurement Conference–IMC*, 2014.

[31] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *ACM Conference on Computer and Communications Security–CCS*, pages 73–84. ACM, 2013.

[32] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer & Communications Security*, 2013.

[33] J. H. Ellis. The story of non-secret encryption, 1997.

[34] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *ACM Conference on Computer and Communications Security – CCS*, pages 50–61. ACM, 2012.

[35] Edward Felten. The Linux backdoor attempt of 2003. Available at `https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003/`, 2013.

[36] Niels Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista, 2006.

[37] Niels Ferguson and Bruce Schneier. A cryptographic evaluation of IPsec. 2003. Available at `https://www.schneier.com/paper-ipsec.html`.

[38] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security – CCS*, 2012.

[39] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, January 1996.

[40] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Usenix Security*, 2008.

[41] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5), May 2009.

[42] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, pages 205–220, 2012.

[43] Lin-Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. In *IEEE Symposium on Security and Privacy*. IEEE, 2014.

[44] Thomas Johnson. American cryptology during the Cold War, 1945–1989. chapter Book III: Retrenchment and Reform, page 232. 2009.

[45] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems–CHES 2009*, pages 1–17. Springer, 2009.

[46] Charlie Kaufman. Differential workfactor cryptography. Available at `http://www.ussrback.com/crypto/nsa/lotus.notes.nsa.backdoor.txt`, 1996.

[47] C.W. Kaufman and S.M. Matyas. Differential work factor cryptography method and system, June 9 1998. US Patent 5,764,772.

[48] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology–CRYPTO*, pages 388–397. Springer, 1999.

[49] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology–CRYPTO*, pages 104–113. Springer, 1996.

[50] Jesse D. Kornblum. Implementing BitLocker drive encryption for forensic analysis. *Digital Investigation*, 5(3):75–84, March 2009.

[51] Venafi Labs. Venafi Labs Q3 Heartbleed threat research analysis, 2014. Available at `https://www.venafi.com/assets/pdf/wp/Venafi_Labs_Q3_Heartbleed_Threat_Research_Analysis.pdf`.

[52] Adam Langley. Enhancing digital certificate security. Available at `http://googleonlinesecurity.blogspot.com/2013/01/enhancing-digital-certificate-security.html`, 2013.

[53] Fredrik Laurin and Calle Froste. Secret Swedish E-mail can be read by the U.S.A. Available at `http://catless.ncl.ac.uk/Risks/19.52.html#subj1`, November 1997.

[54] Nate Lawson. Timing attack in Google Keyczar library. `http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library`, 2009.

[55] George Marsaglia. DIEHARD: a battery of tests of randomness. Available at `http://stat.fsu.edu/~geo/diehard.html`, 1996.

[56] Joseph Menn. Exclusive: Secret contract tied NSA and security industry pioneer. *Reuters*, December 13 2013.

[57] Mohamed Saied Emam Mohamed, Stanislav Bulygin, Michael Zohner, Annelie Heuser, Michael Walter, and Johannes Buchmann. Improved algebraic side-channel attack on AES. *Journal of Cryptographic Engineering*, 3(3):139–156, 2013.

[58] National Institute of Standards and Technology. Special Publication 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators, 2012. Available at `http://csrc.nist.gov/publications/PubsSPs.html#800-90A`.

[59] Nguyen and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3), 2002.

[60] Tyler Nichols, Joe Pletcher, Braden Hollembaek, Adam Bates, Dave Tian, Abdulrahman Alkhelaifi, and Kevin Butler. CertShim: Securing SSL certificate verification through dynamic linking. In *ACM Conference on Computer and Communications Security – CCS*. ACM, 2014.

[61] National Institute of Standards and Technology. Cryptographic standards statement, September 2013.

[62] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA*, pages 1–20. Springer, 2006.

[63] JR Prins and Business Unit Cybercrime. DigiNotar certificate authority breach 'Operation Black Tulip'. *Fox-IT, November*, 2011.

[64] Vincent Rijmen and Bart Preneel. A family of trapdoor ciphers. In *Fast Software Encryption–FSE*, volume 1267 of *LNCS*, pages 139–148. Springer-Verlag, 1997.

[65] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Network and Distributed System Symposium–NDSS*. ISOC, 2010.

[66] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[67] Bruce Schneier. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[68] Bruce Schneier. How to Design—And Defend Against—The Perfect Security Backdoor. `https://www.schneier.com/essays/archives/2013/10/how_to_design_and_de.html`, 2013.

[69] Dan Shumow and Niels Ferguson. On the possibility of a back door in the NIST SP800-90 Dual EC PRNG. Presentation at the CRYPTO 2007 Rump Session, 2007.

[70] Gustavus Simmons. The prisoners' problem and the subliminal channel. In *Advances in Cryptology–CRYPTO*, 1983.

[71] Gustavus Simmons. The subliminal channel and digital signatures. In *Advances in Cryptology–CRYPTO*, 1985.

[72] Gustavus Simmons. Subliminal communication is easy using the DSA. In *Advances in Cryptology–EUROCRYPT*, 1993.

[73] Gustavus Simmons. The history of subliminal channels. In *Information Hiding*. 1996.

[74] Christopher Tarnovsky. Deconstructing a secure processor. BlackHat Briefings, 2010.

[75] Serge Vaudenay. Security flaws induced by CBC padding – applications to SSL, IPSEC, WTLS, ... In *Advances in Cryptolog–EUROCRYPT*. Springer.

[76] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Workshop on Electronic Commerce*, 1996.

[77] Alma Whitten and J. Doug Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium*, volume 1999, 1999.

[78] M.J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36(3), 1990.

[79] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *ACM Internet Measurement Conference–IMC*, pages 15–27. ACM, 2009.

[80] Adam Young and Moti Yung. The dark side of "black-box" cryptography, or: Should we trust Capstone? In *Advances in Cryptology–CRYPTO*, 1996.

[81] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In *Advances in Cryptology–EUROCRYPT*, 1997.

[82] Adam Young and Moti Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In *Advances in Cryptology–CRYPTO*, 1997.

[83] Adam Young and Moti Yung. Monkey: Black-box symmetric ciphers designed for MONopolizing KEYs. In *Fast Software Encryption*. 1998.