Morpheus: Bringing The (PKCS) One To Meet the Oracle

Moosa Yahyazadeh
The University of Iowa
moosa-yahyazadeh@uiowa.edu

Sze Yiu Chau
The Chinese University of Hong Kong
sychau@ie.cuhk.edu.hk

Li Li
Syracuse University
lli101@syr.edu

Man Hong Hue
The Chinese University of Hong Kong
manhonghue@cuhk.edu.hk

Joyanta Debnath
The University of Iowa
joyanta-debnath@uiowa.edu

Sheung Chiu Ip
The Chinese University of Hong Kong
ipsheungchiu@gmail.com

Chun Ngai Li
The Chinese University of Hong Kong
1155110647@link.cuhk.edu.hk

Endadul Hoque
Syracuse University
enhoque@syr.edu

Omar Chowdhury
The University of Iowa
omar-chowdhury@uiowa.edu

ABSTRACT

This paper focuses on developing an automatic, black-box testing approach called Morpheus to check the non-compliance of libraries implementing PKCS#1-v1.5 signature verification with the PKCS#1-v1.5 standard. Non-compliance can not only make implementations vulnerable to Bleichenbacher-style RSA signature forgery attacks but also can induce interoperability issues. For checking non-compliance, Morpheus adaptively generates interesting test cases and then takes advantage of an oracle, a formally proven correct implementation of PKCS#1-v1.5 signature standard, to detect non-compliance in an implementation under test. We have used Morpheus to test 45 implementations of PKCS#1-v1.5 signature verification and discovered that 6 of them are susceptible to variants of the Bleichenbacher-style low public exponent RSA signature forgery attack, 1 implementation has a buffer overflow, 33 implementations have incompatibility issues, and 8 implementations have minor leniencies. Our findings have been responsibly disclosed and positively acknowledged by the developers.

CCS CONCEPTS

• **Security and privacy → Digital signatures**; **Formal security models**; **Logic and verification**; *Security protocols.*

KEYWORDS

PKCS#1 signature verification; non-compliance checking; reference implementation; adaptive combinatorial testing

ACM Reference Format:
Moosa Yahyazadeh, Sze Yiu Chau, Li Li, Man Hong Hue, Joyanta Debnath, Sheung Chiu Ip, Chun Ngai Li, Endadul Hoque, and Omar Chowdhury. 2021. Morpheus: Bringing The (PKCS) One To Meet the Oracle. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3460120.3485382

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3485382

1 INTRODUCTION

RSA digital signature scheme is a fundamental cryptographic primitive that enjoys a widespread deployment in many different application domains including secure communication protocols (e.g., SSL/TLS, IPSec), software signing, X.509 certificates, to name a few. As an example, 95% of the X.509 certificates in the Censys dataset [2] use RSA digital signatures. A common security requirement for digital signature schemes is the (*existentially*) *unforgeable under chosen-message attacks* property which we write in short as resistant to existential forgery (REF) property. Realizing such guarantees with the RSA digital signature scheme at implementation level requires carefully padding the hash digest of the message to be signed using a secure padding scheme [40, 56].

One of the most prominent among these padding schemes is the PKCS#1-v1.5 [45–47, 56]. Although some security properties of PKCS#1-v1.5 signature scheme specification has been formally proved [40], a significant number of its implementations have been shown to suffer from some critical vulnerabilities [25, 33, 38, 39, 53]. The most severe among these implementation flaws have been shown to enable an attacker to forge a signature *without the knowledge of the private key*, especially when a small public exponent is used. These attacks are variants of the Bleichenbacher-style low public exponent RSA signature forgery [38] that have been around for more than a decade. . Note that, the attacks on RSA signature verification considered here is different from *the padding oracle attack*, also attributed to Daniel Bleichenbacher [26], which exploits RSA private-key operations and can manifest via various side channels [27, 41, 50, 55, 59]. With RSA signatures being implemented in many different languages and platforms, implementation flaws causing violations of the REF property can be catastrophic. It is thus paramount to develop an approach to check an implementation's non-compliance to the standard [45–47, 56]. Non-compliant implementations can not only suffer from the violation of the REF property but also induce interoperability issues.

Prior efforts that analyzed PKCS#1-v1.5 implementations have been mostly manual [38, 53]. Recently, Chau *et al.* [33] developed an approach based on symbolic execution for testing non-compliance of PKCS#1-v1.5 implementations. However, due to the need for intrusive source-code level changes and toolchain limitations (especially for programming languages that lack a mature LLVM front-end to translate programs into the subset of LLVM IR supported by KLEE), their study only considered open source C implementation, leaving a large landscape of PKCS#1-v1.5 implementations written in other programming languages untested and potentially vulnerable. To make matters worse, PKCS#1-v1.5 have also been implemented in embedded devices for which one only has black-box access, further complicating non-compliance checking. *To improve the unsatisfactory state of affairs, in this paper, we set out to develop an automated, black-box non-compliance checker called Morpheus.* Performing non-compliance checking in a black-box fashion makes Morpheus agnostic to the subject implementations' programming languages, and thus enables it to cover a diverse set of PKCS#1-v1.5 libraries, many of which were not studied before.

As capturing the PKCS#1-v1.5 standard requirements warrants a context-sensitive grammar, the underlying non-compliance checking problem that Morpheus addresses can be stated as follows: *Given black-box access to a PKCS#1-v1.5 signature verification implementation I and the PKCS#1-v1.5 standard requirements represented as a context-sensitive grammar Γ, is the grammar implemented by I inequivalent to Γ?* Unfortunately, this is an undecidable problem even when the grammar implemented by I is given.

Proposed approach (Morpheus). Morpheus addresses this problem through the introduction of two components, namely, the *input sampler* and the *oracle*. At a high level, to test implementation I, Morpheus's input sampler keeps selecting concrete inputs from the input space, and feeds them to both I and Morpheus's oracle. If responses from I and the oracle differ, the concrete input in question is then an evidence of I being non-compliant. One can instantiate the input sampler with any (grammar-based) fuzzer but due to the lack of feedback information under the black-box settings, we observed that the fuzzer-based input sampler was having limited success in identifying non-compliance instances. To address this, we developed a specialized input sampler, based on our domain knowledge, which intelligently samples the input space in an adaptive fashion. This is similar to an adaptive, combinatorial testing approach in which once a non-compliance (synonymously, leniency in the implementation) is discovered by a generated test case, it adaptively generates more test cases of the same class to reveal if the leniency in the subject implementation can be exploited for Bleichenbacher-style low public exponent RSA signature forgery.

We instantiate Morpheus's oracle with a formally proven correct implementation of the PKCS#1-v1.5 standard. We implement the oracle in Gallina [34] and use Coq's interactive theorem prover to verify that the implementation complies with the requirements of the PKCS#1-v1.5 standard. We then use Coq's extraction mechanism to obtain an OCaml source code.

Findings. To show the efficacy of Morpheus, we analyzed the recent versions of 45 PKCS#1-v1.5 signature verification implementations, written in 18 different programming languages. We have discovered that 40 of these libraries are non-compliant with the standard. Among them, 6 implementations have leniencies leaving significant areas of the PKCS#1-v1.5 encoded message structure unchecked, enabling an attacker to launch the Bleichenbacher-style low public exponent RSA signature forgery attack. For some PKCS#1-v1.5 libraries (e.g., node-forge), the size of unchecked area is so large that the Bleichenbacher-style low exponent signature forgery, typically possible for $e = 3$, become practical even for $e = 5$ or $e = 17$.

Although $e = 3$ is seldom used currently by certificates on the Internet [36], we note that small public exponents are not yet extinct. From the Censys [2] certificates dataset (2019 snapshot) containing a total of 1,234,185,668 certificates, we found that 0.07% has $e = 3$ and 0.14% has $e = 17$ as their RSA public exponents. More importantly, many Linux distributions continue to have some trusted root CA certificates with $e = 3$ in their default CA bundle. $e = 3$ is also sometimes mandated by key generation programs [5] and has been historically recommended for better performance [37], especially in resource-constrained devices. Also, mathematically it is interesting to see how such leniency enable attackers to target slightly larger public exponents, and how the forgery attack is counter-intuitively easier to succeed under a choice of parameter that is supposed to improve security (*i.e.*, using a longer RSA modulus).

We have found other minor leniencies in 8 libraries that cause some invalid signatures to be mistakenly accepted as valid signatures. Although these leniencies do not directly lead to signature forgeries, there are some that contribute towards the success of attacks when exploited in tandem. Besides these semantic correctness issues in the lenient libraries' signature verification logics, we have also discovered 1 buffer overflow vulnerability in `Relic`. Our results also show that 33 libraries have incompatibility issue where a valid signature, whose encoded message uses implicit NULL for hash algorithm parameter, is being mistakenly rejected, which can create an interoperability issue. Based on a random sampling performed on Censys certificates dataset, we observed 4% of the certificates use implicit NULL parameter in their signatures.

Responsible disclosure. Following the practice of responsible disclosure, we have notified all of the vendors mentioned in this paper about our findings. The vendors have also acknowledged our findings and 12 CVEs have been assigned. Furthermore, we have participated in the design and/or verification of the patches.

Contributions. This paper makes the following contributions:

(1) We have implemented a formally verified PKCS#1-v1.5 signature verification that can be used as the oracle to perform the non-compliance checking of PKCS#1-v1.5 libraries.
(2) We propose an adaptive combinatorial testing approach (Morpheus) to generate effective test cases — adhering to PKCS#1-v1.5 signature scheme's context-sensitive grammar — in order to do non-compliance checking of a diverse set of PKCS#1-v1.5 libraries in a black box fashion.
(3) We evaluated Morpheus by analyzing 45 PKCS#1-v1.5 libraries and discovered 6 of them to suffer from Bleichenbacher-style low exponent signature forgery, along with other bugs. We responsibly disclosed our findings to the affected vendors, and contributed in the design and/or testing of the proposed patches. Our findings are also accompanied by theoretical analysis and proof-of-concept attacks in Section 6.

2 PRELIMINARIES

The PKCS#1-v1.5 signature scheme delineates signature generation (*i.e.*, sign) operation and signature verification (*i.e.*, verify) operation. We use the notation in Table 1 as a reference throughout the paper.

Signature generation operation. Given d, m, and H as input, the signature generation operation outputs S as follows. It first computes the hash value of m based on the given hash function H

(*e.g.*, SHA-256). It then encodes the algorithm ID of the used hash function and the hash value into the DER (Distinguished Encoding Rules [10]) encoded ASN.1 value of the DigestInfo type.

```
DigestInfo ::= SEQUENCE {
        digestAlgorithm AlgorithmIdentifier,
        digest OCTET STRING
}
```

in which digestAlgorithm is the structure where the algorithm ID resides and digest is the structure for the hash value. Let ASN.1 DER encoding of the DigestInfo be T, it then forms the encoded message, EM, by concatenating these bytes as follows:

```
EM = 0x00 || BT || PS || 0x00 || T
```

where 0x00 is the value for leading byte, BT for signature scheme is 0x01, PS is padding string at least 8 octets long with hexadecimal value 0xFF for each padding byte, ending with a byte with value 0x00 showing the end of padding. All these bytes are considered as a prefix to T, and the length of PS is computed as $|PS| = |n| - |T| - 3$ to make the length of the encoded message equal to the length of the public modulus (*i.e.*, $|EM| = |n|$). Once EM is formed, the signature S is computed as $S = EM^d \ mod \ n$.

Signature verification operation. Given (n, e), m_v, S, and H, the signature verification operation verifies whether S is a valid signature. For this, it first checks if the length of the signature S is equal to $|n|$; otherwise, it returns "invalid signature". Then, it obtains the encoded message from S using $EM_v = S^e \ mod \ n$. After this, it can follow one of the following approaches to complete verification.

Encoding approach: In this approach (a.k.a., construction-based approach), the verifier first computes $H(m_v)$ and uses it to construct the ASN.1 DER encoding of DigestInfo to eventually form its version of the encoded message, EM'_v, following the same approach as the signer. Once EM'_v is constructed, the verifier checks $EM'_v \overset{?}{=} EM_v$. In case they are equal, the verifier outputs "valid signature". Otherwise, "invalid signature" is returned.

Decoding approach: In this approach (a.k.a., parsing-based approach), the verifier parses EM_v and strips off the prefix bytes to obtain the hash value and digestAlgorithm structure from T. It then checks that the digestAlgorithm to be consistent with the given H, and once passed, the verifier computes its own version of the hash value of the given message (*i.e.*, $H(m_v)$). Finally, the computed hash value is compared against the obtained hash value (from the encoded message EM_v). If they are equal, the verifier outputs "valid signature". Otherwise, it returns "invalid signature".

To NULL or not to NULL. The PKCS#1-v1.5 standard has evolved through multiple RFCs (see Appendix A), and a point of confusion concerns the parameter field of the hash algorithm meta-data encoded in the DER format. As the SHA family of hash functions do not need a parameter, the question then becomes how should this NULL parameter be encoded. The initial recommendation was that the parameter field be absent for such hash functions; denoting an *implicit* NULL parameter. The recommendation was later changed to have the parameter field be present with an explicit NULL. Recently, this recommendation has been updated to allow both explicit and implicit NULL values [56]. As discussed in Section 6, confusions over this has led to some incompatibility issues.

Table 1: Reference notation

Symbol	Description
n	RSA public modulus
$\|n\|$	Length of RSA public modulus
e	RSA public exponent
(n, e)	The signer's public key
d	RSA private exponent
m	Message to be signed
m_v	Message received by verifier
EM	Encoded message, an octet string input to the sign operation
S	Signature, an octet string computed as $EM^d \ mod \ n$ by signer
EM_v	Encoded message, an octet string computed as $S^e \ mod \ n$ by verifier
H	Hash function
$H(m_s)$	Sign operation version of H(m), contained inside EM_v
$H(m_v)$	Verify operation's computed hash of m_v
EM'_v	Encoded message, an octet string constructed by verify operation
BT	Block type byte
PS	Padding string
T	ASN.1 DER encoding of the DigestInfo value

3 OVERVIEW OF MORPHEUS

In this section, we discuss the problem definition Morpheus addresses along with its high-level approach.

3.1 Problem Definition

In this paper, we focus on checking whether a given implementation (I) of PKCS#1-v1.5 scheme is *non-compliant* with its standard. The requirements of PKCS#1-v1.5 standard can be represented as a context-sensitive grammar Γ. This is due to the context-sensitivity needed to capture the requirements associated with the PS (padding string) and T (ASN.1 DER encoded DigestInfo) fields of the encoded message EM. Based on this, we can define our problem as follows: *Given a context-sensitive grammar Γ capturing the requirements of the PKCS#1-v1.5 standard and given only black-box access to an implementation I of the PKCS#1-v1.5 scheme, is I non-compliant with* Γ? We say I is *non-compliant* with the standard Γ if and only if there exists an input x such that one of the following holds: $\mathbf{NC_1}$) $x \notin \Gamma$ but $I(x) = true$; $\mathbf{NC_2}$) $x \in \Gamma$ but $I(x) = false$. We write $x \in \Gamma$ (resp., $x \notin \Gamma$) to denote that x is accepted (resp., rejected) by the grammar Γ. Similarly, an implementation I of PKCS#1-v1.5 scheme returns true for a given input x if and only if x is legitimate according to I.

In the non-compliance checking problem, we are only given black-box access to I. Even if we were given access to the underlying context-sensitive grammar $\Gamma^\star$ that I implements, checking non-compliance with the standard Γ would entail checking the non-emptiness of the following two sets: (a) $\Gamma^\star \setminus \Gamma$; (b) $\Gamma \setminus \Gamma^\star$. As checking non-emptiness of a context-sensitive grammar is an undecidable problem, checking non-compliance problem is also undecidable.

3.2 Non-compliance and Security

Non-compliance to the PKCS#1-v1.5 scheme may result in breaking the security of a RSA signature scheme. There are basically four classes of attacks applicable to any digital signature scheme [63, 64], which are listed as follows from the strongest to the weakest security assumptions: (i) *total break*; (ii) *universal forgery*; (iii) *selective forgery*; (iv) *existential forgery*. A signature scheme is deemed to have the strongest security requirement, if it is secure against the weakest attack (*i.e.*, existential forgery). The strongest security requirement of a signature scheme is thus *Resistance to Existential Forgery* (**REF**), which ensures that for a secure signature scheme there does not exist a message for which an attacker can forge a legitimate signature without knowing the private key.

We now discuss which types of non-compliance to the PKCS#1-v1.5 standard can result in an insecure RSA digital signature scheme implementation (*i.e.*, breaking REF). In what follows, we use I to denote the implementation-under-test of PKCS#1-v1.5 standard, $\mathcal{S}$ to denote the universe of all signatures, and Γ to denote the context-sensitive grammar capturing the PKCS#1-v1.5 standard requirements. We characterize the non-compliance into the following 3 sets, and show their relationships to REF security property.

Leniency Set (L_I). This set contains signatures that I mistakenly accepts but is rejected by Γ. Concretely, $L_I = \{s \mid s \in \mathcal{S} \wedge s \notin \Gamma \wedge I(s) = \mathit{true}\}$.

Forgery Set (F_I). This is the set of signatures that I mistakenly accepts (albeit, rejected by Γ) and causes I to be susceptible to existential forgery attack. Clearly, we have $F_I \subseteq L_I$.

Overly-Restrictive Set (OR_I). This is the set of signatures that I mistakenly rejects whereas it is accepted by Γ. Concretely, $OR_I = \{s \mid s \in \mathcal{S} \wedge s \in \Gamma \wedge I(s) = \mathit{false}\}$ and $OR_I \cap L_I = \emptyset$.

Operationally, we say I is *non-compliant to the standard* Γ *iff* $L_I \neq \emptyset$ or $OR_I \neq \emptyset$. A given PKCS#1-v1.5 implementation-under-test I *has the REF property iff* the set F_I for I is empty (*i.e.*, $F_I = \emptyset$).

3.3 High-Level Approach of Morpheus

For checking non-compliance of a PKCS#1-v1.5 implementation I with the standard Γ, Morpheus tries to find signatures $s \in \mathcal{S}$ such that $s \in L_I$, $s \in F_I$, or $s \in OR_I$. As the cardinality of $\mathcal{S}$ can be large (*i.e.*, 2^{2048} for a 2048-bit RSA exponent), exhaustively enumerating the space $\mathcal{S}$ is infeasible.

For finding non-compliance instances, Morpheus uses an *input sampler* that samples the space $\mathcal{S}$ intelligently to find signatures to test a given implementation I. To check whether a given signature $s \overset{?}{\in} L_I$ or $s \overset{?}{\in} OR_I$, we need a representation of the standard Γ. For Γ, Morpheus uses a formally verified implementation of PKCS#1-v1.5 standard I_{oracle} (*oracle*) as a proxy to test whether $s \overset{?}{\in} \Gamma$ during non-compliance checking (i.e., $I_{\text{oracle}}(s) = \mathit{true} \Leftrightarrow s \in \Gamma$).

4 THE ORACLE OF MORPHEUS

Morpheus's oracle is a formally proven correct implementation of PKCS#1-v1.5 signature verification standard. Developing the oracle involves four main steps: (1) consulting the standard to extract the specifications and formalizing them as the correctness criteria; (2) developing the actual implementation of PKCS#1-v1.5 signature verification function; (3) proving that the implementation satisfies the specification using an interactive theorem prover (ITP); and finally (4) extracting an executable binary of the oracle to be used as a reference implementation for further non-compliance checking purposes. We use Coq [34] as the interactive theorem prover.

4.1 Formalizing the Specifications

From the English descriptions provided in the RFC8017 standard, we have formalized the specifications of the signature verification in Coq's specification language, Gallina. The formal specification serves as the correctness criteria for the signature verification function. Using the reference notation in Table 1, the original inputs to PKCS#1-v1.5 signature verification are S, m_v, H, and (n, e). Separating cryptographic primitives (such as hash function operation and modular exponentiation) from signature verification operation, we can invest our efforts to the verifier's logic itself. We design the interface so that instead of the signature value S, the oracle accepts the encoded message EM_v (as a buffer calculated by $S^e \bmod n$). For the next argument, instead of the message m_v, it takes the computed hash value $h_v = H(m_v)$. Since all cryptographic operations are performed outside this verification, we can replace the signer's public key (n, e) in the inputs with the length of public modulus n, denoted by n_l. Based on that, we can describe Signature_verification's correctness using the following theorem:

Theorem 4.1 [Signature verification correctness]

$$\begin{aligned}
&\forall\ (EM_v,\ h_v : \mathit{list\ byte})(n_l : \mathit{nat})(H : \mathit{hash\_id}),\\
&\quad((\textsc{Signature\_verification}\ EM_v\ n_l\ h_v\ H)\ =\ \mathit{true}) \Longleftrightarrow\\
&\quad((\textsc{Pow}\ 2\ (\textsc{Log2}\ n_l) = n_l) \wedge\\
&\qquad((\textsc{Length}\ EM_v) = n_l) \wedge\\
&\qquad((\textsc{Length}\ h_v) = (\textsc{H2Len}\ H)) \wedge\\
&\qquad(EM_v[0] = \mathtt{0x00}) \wedge\\
&\qquad(EM_v[1] = \mathtt{0x01}) \wedge\\
&\qquad(\exists\ (l_1,\ l_2 : \mathit{nat})(a_1,\ a_2 : \mathit{asb}),\\
&\qquad\quad((\textsc{H2Asn}\ H\ h_v) = \langle\langle a_1,\ l_1\rangle,\ \langle a_2,\ l_2\rangle\rangle) \wedge\\
&\qquad\quad(\textsc{Asb\_is\_valid}\ a_1)\ \wedge (\textsc{Asb\_is\_valid}\ a_2) \wedge\\
&\qquad\quad((n_l - l_1) - 3 >= 8)\ \wedge ((n_l - l_2) - 3 >= 8) \wedge\\
&\qquad\quad(((\forall\ (i : \mathit{nat}),\ ((i >= 2)\ \wedge\ (i < ((n_l - l_1) - 1))) \Longrightarrow\\
&\qquad\qquad(O[i] = \mathtt{0xFF})) \wedge\\
&\qquad\qquad(EM_v[((n_l - l_1) - 1)] = \mathtt{0x00}) \wedge\\
&\qquad\qquad(\forall\ (j : \mathit{nat}), ((j >= 0)\ \wedge\ (j < l_1)) \Longrightarrow\\
&\qquad\qquad\quad(EM_v[((n_l - l_1) + j)] = (\textsc{Asb2Byte}\ a_1)[j]))\\
&\qquad\quad) \vee\\
&\qquad\quad((\forall(i : \mathit{nat}),\ ((i >= 2); \wedge\ (i < ((n_l - l_2) - 1))) \Longrightarrow\\
&\qquad\qquad(EM_v[i] = \mathtt{0xFF})) \wedge\\
&\qquad\qquad(EM_v[((n_l - l_2) - 1)] = \mathtt{0x00}) \wedge\\
&\qquad\qquad(\forall(j : \mathit{nat}),\ ((j >= 0); \wedge\ (j < l_2)) \Longrightarrow\\
&\qquad\qquad\quad(EM_v[((n_l - l_2) + j)] = (\textsc{Asb2Byte}\ a2)[j]))))))
\end{aligned}$$

where $n_l = |\mathtt{n}|$, $h_v = H(m_v)$, and H is an enumeration denoting the hash function whose element is drawn from the set $\mathit{hash\_id} = \{$ `SHA-1`, `SHA-224`, `SHA-256`, `SHA-384`, `SHA-512`$\}$. Here, (Pow x y) is the power function that raises x to y whereas (Log2 x) is the base-2 logarithm function. We have (Length x) function that takes a list as input and returns its length. (H2Len x) function is defined such that given a hash function ID as input, it returns the length of hash value of to-be-signed message. Also, *asb* is the type for ASN.1 DER encoded portion of the signature scheme structure and (H2Asn x y) is a function that given x as a hash function ID and y as a hash value byte list, it returns a pair of pairs $\langle\langle a_1,\ l_1\rangle,\ \langle a_2,\ l_2\rangle\rangle$ such that a_1 is the constructed ASN.1 DER encoded bytes of type *asb* which contains the given hash function ID and hash value byte list with explicit hash algorithm parameter; and l_1 is a_1's length, while a_2 and l_2 are for the implicit NULL algorithm parameter counterpart. (Asb_Valid x) is a function that checks whether the given *asb* structure, x, is valid (*i.e.*, it conforms to the ASN.1 DER encoded bytes of the signature scheme). Finally, (Asb2Byte x) takes *asb* typed value, x, and returns its serialized version.

Basically, Theorem 4.1 has two parts that express the following:

(1) If the oracle returns *true* for any input, then that input is valid according to the formal specifications (*i.e.* ***soundness***);
(2) For all inputs accepted by the specification, the oracle must return *true* (*i.e.* ***completeness***).

4.2 Developing the PKCS#1 Implementation

Our oracle follows the construction-based approach where two versions of the encoded message are constructed (for both explicit and implicit NULL parameter cases) where they are compared for equality against the input encoded message (see Algorithm 1).

Concretely, our oracle named Signature_verification first checks whether n_l is the power of two. It also checks that the EM_v's length is equal to the public modulus length n_l. It then checks that the length of hash value matches with a given hash function's output length. If these checks fail, then the oracle returns $false$; otherwise, it builds the ASN.1 DER structure from the given hash function H and the hash value h_v for both explicit and implicit NULL parameter cases using the H2Asn function. Signature_verification then constructs two PKCS#1-v1.5 signature scheme structures by adding the leading byte, block type byte, padding bytes, and the end of padding to the both ASN.1 DER versions.

The necessary number of padding bytes is computed as $(n_l - l) - 3$ for those two PKCS structures, where l is replaced with the ASN.1 sub-structure's length for each version (*i.e.*, l_1 for explicit version and l_2 for implicit version). Once the PKCS structures are constructed, they are checked for validity with Pkcs_format_is_valid (Algorithm 2). This function verifies that all components in PKCS structure have a correct value and a correct length. If this check fails, Signature_verification returns $false$; otherwise, it converts both PKCS structures (for explicit and implicit formats) into the list of bytes (via Pkcs_format_to_byte function) and then it checks the equality between the PKCS converted byte list and the given encoded message EM_v. Only when EM_v matches with either of the explicit or implicit PKCS byte lists, *true* is returned.

4.3 Proof Sketch

Our proof proceeds by proving both soundness and completeness parts of the correctness theorem. We provide a proof sketch here.

Soundness (*i.e.*, first direction ⇒): (1) We assume (Signature_verification EM_v n_l h_v H) = *true*, then we prove the consequent of the logical implication; (2) By unfolding Signature_verification definition from Algorithm 1 in our hypotheses section, we can destruct the condition of the first **if** and then prove:

$$((\text{Pow } 2\ (\text{Log2 } n_l) = n_l) \quad \land \quad ((\text{Length } EM_v) = n_l) \quad \land \quad ((\text{Length } h_v) = (\text{H2Len } H)))$$

(3) Continuing down the first **if**'s true branch, and having validated the constructed PKCS format given the Pkcs_format_is_valid algorithm in 2, we get into the second **if**'s true branch where the constructed PKCS format is converted into byte list. No matter which versions of structure it is (*i.e.*, structure for explicit or implicit versions), we can prove that the first and second elements in the lists are `0x00` and `0x01`, respectively, based on the PKCS construction and the subsequent serialization; hence, we can prove:

$$(EM_v[0] = \texttt{0x00}) \ \land (EM_v[1] = \texttt{0x01})$$

(4) Since the two versions of PKCS structure are validated in the second **if**'s true branch, and those contain the ASN.1 sub-structures generated by H2Asn, then the ASN.1 sub-structures must be valid. That is, having Pkcs_format_is_valid for a PKCS structure implies Asb_is_valid for the engulfing ASN.1 sub-structure (see algorithm 2). Therefore, we have the proofs for:

$$(\text{Asb\_is\_valid } a_1) \ \land (\text{Asb\_is\_valid } a_2)$$

(5) For a valid PKCS format, it has been checked that the padding bytes are at least 8. It is done by Padding_bytes_length_ge_8_and_all_ff function inside the algorithm 2, Pkcs_format_is_valid. Therefore, if we just subtract the lengths of ASN.1 sub-structure and three bytes from the length of PKCS structure (for leading byte, block type byte and padding end byte), we will end up with the length of padding bytes block which is at least 8 bytes. Therefore, we have proved:

$$((n_l - l_1) - 3 >= 8) \ \land ((n_l - l_2) - 3 >= 8)$$

Algorithm 1 Signature_verification's definition

```
Definition signature_verification ($EM_v$ : list byte) ($n_l$ : nat) ($h_v$ : list byte) ($H$ : hash_id) : bool :=
  if (((Is_power_of_two $n_l$) &&
      ((Length $EM_v$) =? $n_l$)) &&
      ((Length $h_v$) =? (H2Len $H$))) then
    match (H2Asn $H$ $h_v$) with
    | alp $a_1$ $l_1$ $a_2$ $l_2$ ⇒
      if (Pkcs_format_is_valid
          (pkcs (0x00) (0x01) (Repeat 0xFF (($n_l$ - $l_1$) - 3)) (0x00) $a_1$))
        && (Pkcs_format_is_valid
          (pkcs (0x00) (0x01) (Repeat 0xFF (($n_l$ - $l_2$) - 3)) (0x00) $a_2$))
      then
        (List_eq $EM_v$ (Pkcs_format_to_byte
         (pkcs (0x00) (0x01) (Repeat 0xFF (($n_l$ - $l_1$) - 3)) (0x00) $a_1$))
        ||
        List_eq $EM_v$ (Pkcs_format_to_byte
         (pkcs (0x00) (0x01) (Repeat 0xFF (($n_l$ - $l_2$) - 3)) (0x00) $a_2$)))
      else false
    end
  else false.
```

Algorithm 2 Pkcs_format_is_valid's definition in Coq

```
Definition Pkcs_format_is_valid (st : pkcs_format) : bool :=
 match st with
 | pkcs leading_byte block_type_byte padding_bytes padding_end asn_block

   ⇒ ((Byte.eqb leading_byte 0x00) &&
      (Byte.eqb block_type_byte 0x01) &&
      (Padding_bytes_length_ge_8_and_all_ff padding_bytes) &&
      (Byte.eqb padding_end 0x00) &&
      (Asb_is_valid asn_block))
 end.
```

(6) Given the List_eq's correctness as well as being in the second **If**'s true branch from the initial assumption, we know that one of the List_eq function calls in the disjunction must be *true*. Therefore, EM_v must be equal to one of the PKCS structures.
$\therefore$ first direction ($\Rightarrow$) is proved.

Completeness (*i.e.*, second direction $\Leftarrow$): (1) We assume the right portion of the equivalence to be true, then we need to prove (Signature_verification EM_v n_l h_v H) = *true*; (2) Given the following conjunction in our hypotheses section,

$$((\text{Pow } 2\ (\text{Log2 } n_l) = n_l) \quad \wedge \quad ((\text{Length } EM_v) = n_l) \\ \wedge \quad ((\text{Length } h_v) = (\text{H2Len } H)))$$

we can prove that the conditions of the first **if** statement in Signature_verification definition are *true*; (3) Then, using the following lemma,
Lemma *ifT* : $b \Rightarrow$ (**if** b **then** *true* **else** *false*) = *true*.
we rewrite the second **if**'s expression. Now we have to prove the body of the second **if** is true as well as its assumed condition expression; (4) The body of the second **if** has a disjunction; so, we have to prove them one at a time. For the left side of the disjunction, we unfold the definition of Pkcs_format_to_byte which after simplification, we get the following expression:

$$(\text{List\_eq}\ \ EM_v\ \ [\texttt{0x00}; \texttt{0x01}] + + (\text{Repeat } \texttt{0xFF}\ (n - l_1 - 3)) \\ + + [\texttt{0x00}] + + (\text{Asb\_to\_byte } a_1))$$

(5) On the other hand, we have in our hypotheses a specification expressing a list of byte. So, we use the following Lemma:

Lemma *build_from_spec* :
$\forall$ (a_{len} b_{len} : *nat*) (a b : *list byte*),
(((($Datatypes.length\ a = a_{len}$ $\wedge$
$Datatypes.length\ b = b_{len}$) $\wedge$
$7 < a_{len} - b_{len} - 3$) $\wedge$
$a[0]$ = `0x00`) $\wedge$
$a[1]$ = `0x01`) $\wedge$
($\forall$ i : *nat*, $(1 < i) \wedge (i < a_{len} - b_{len} - 1) \Rightarrow a[i]$ = `0xFF`) $\wedge$
$a[(a_{len} - b_{len} - 1)]$ = `0x00` $\wedge$
($\forall$ j : *nat*, $(0 \leq j) \wedge (j < b_{len}) \Rightarrow a[(a_{len} - b_{len} + j)] = b[i]$
$\Rightarrow$
a = [`0x00`;`0x01`] ++ (Repeat `0xFF` ($a_{len} - b_{len}$ - 3)) ++ [`0x00`] ++ b.

to apply it on the specification we have in the hypotheses section. Now we can rewrite the expression we have in our proof section with the one we have achieved by applying the above lemma, and thus get the following expression: (List_eq EM_v EM_v), which by simplification and List_eq_correcness lemma we have it to be *true*; (6) Same steps also apply to the right side of the disjunction as above and $\therefore$ we get the proof completed. The full formal proof in Coq and other resources can be found in [20].

4.4 Extraction

After proving Theorem 4.1, we extract an OCaml program out of the Gallina description of the oracle using Coq's built-in extraction feature. We put the OCaml program inside a wrapper code that handles the command line arguments. Finally, the OCaml code will be compiled to achieve the oracle's executable binary.

5 TESTING WITH MORPHEUS

We now describe the concrete approach taken by Morpheus to find noncompliance in a PKCS#1-v1.5 signature verifier.

5.1 Architecture of Morpheus

Morpheus comprises of two main components, namely, the *input sampler* and *bug detector* (See Figure 1). The input sampler samples the possible input space of PKCS#1-v1.5 encoded messages and selects interesting concrete inputs, possibly using feedback from the bug detector. These concrete inputs are then fed into the bug detector. The bug detector takes these inputs, then does the following for each input in parallel: (1) it pre-processes the input; (2) feeds the input to both the oracle and implementation-under-test; (3) compares their outputs, and if there is a discrepancy, then reports the input as an evidence of non-compliance of the test subject; (4) finally, it reports back the output comparison status as a feedback to the input sampler. The pre-processing is done to take into account the difference between the interfaces of the oracle and the test subject. The status of the output comparison is used adaptively by the input sampler for generating more inputs of a particular class when a non-compliance has been detected. We now provide more details on the Morpheus's input sampler component as the functionality of the bug detector has been discussed already.

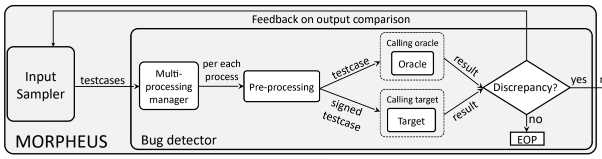

Figure 1: The high-level architecture of Morpheus

5.2 Insight of Morpheus's Input Sampler

One may question the rationale of designing a custom input sampler for Morpheus instead of using a mutation engine from an existing fuzzer. As we will demonstrate in Section 6.2.1, due to the lack of domain-specific knowledge, existing mutation approaches fail to achieve the same level of proficiency in identifying non-compliance instances compared to Morpheus's input sampler.

The input sampler we design follows the *adaptive combinatorial testing approach*, which takes the union of two different test generation strategies, namely, *combinatorial testing* and *adaptive domain-specific test generation strategy*. Precisely, Morpheus individually generates inputs with each of these approaches and then unify them into a single set of test cases. For both these approaches, we decompose the PKCS#1-v1.5 encoded message (i.e., EM_v) into different components (e.g., `BT`, `PS`).

On one hand, combinatorial testing is a general testing method to verify interactions among test factors. It has been traditionally used for software testing with the goal of reducing the cost of test case generation while maintaining the effectiveness, with the key insight that not every input parameter contributes to every failure [28, 35, 51, 61, 62]. In our context, each of the components serve as a test factor in our test generation. Precisely, for each of the components, we have a set of interesting byte sequences, chosen based on the neighboring components, that we can use to test an implementation. We combinatorially combine these different interesting bytes sequences for each component to generate test cases. Note that, we do not shuffle the order of the components as doing so generally destroys the structure and are rightly rejected by most implementations. On the other hand, the adaptive domain-specific test generation strategy as the name suggests uses domain-specific knowledge about PKCS#1-v1.5 and the Bleichenbacher-style low public exponent RSA signature forgery. It particularly leverages the notion of *byte stealing* and *byte hiding*. Without loss of generality, we will explain these two concepts using an example. Suppose an implementation does not check the well-formedness (e.g., the correct length) of two components c_1 and c_2, each of which is 8-bytes long. In this case, we can possibly steal a byte from c_1 and hide it in c_2. Concretely, we have a test case in which the length of c_1 is 7-bytes whereas c_2 is 9-bytes long. The amount of bytes one can steal should be equal to the amount of bytes one can hide to make sure that the resulting message is always of length n (i.e., the RSA modulus size). As discussed in section 6, stealing and hiding bytes allow an attacker to possibly launch signature forgery attacks on an implementation. The more bytes one can hide, the higher the chances of such an attack. The adaptive nature of this test generation comes into play when the input generator observes as feedback that it was able to steal a byte from one component and hide it in another component. It then adaptively tries to steal and hide more and more bytes.

5.3 Component Decomposition

A PKCS#1-v1.5 encoded message has the following structure depicted in Figure 2. The length of padding bytes is $8 + L_{rp}$ where L_{rp} is the length of the rest of padding bytes, excluding the minimum required 8 bytes. The ASN.1 DER component's length is denoted by L_{asn} variable and its value is denoted using ψ representing a sequence of bytes. Given that we have $|n| = 1+1+8+L_{rp}+1+L_{asn}$.

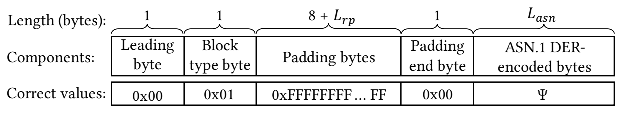

Figure 2: PKCS#1 v1.5 encoded message structure

A closer look at the ASN.1 DER component brings us to the internal structures illustrated in Figure 3, for the case it uses the explicit hash algorithm parameter. Besides the correlations within TLV triplets (*i.e.*, $\langle Type, Length, Value\rangle$), there is a relation between the value of hash ID value component and the length of hash value's value. For example, if we use `SHA-256` as the hash function, then HID = `0x608648016503040201` (which is an object identifier for `SHA-256`) and since `SHA-256` produces a 256-bit (32 bytes) hash value, the length of the hash value's content octet must be 32 bytes (*i.e.*, L_{hvv} = `0x20`). Understanding this internal structure, we can also conclude $L_{asn} = L_{asb} + 2$.

By flattening ASN.1 DER structure, we can decompose an encoded message EM_v structure into 17 components, which are then

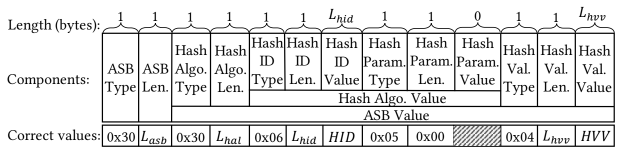

Figure 3: ASN.1 DER encoded structure with explicit hash algorithm parameter

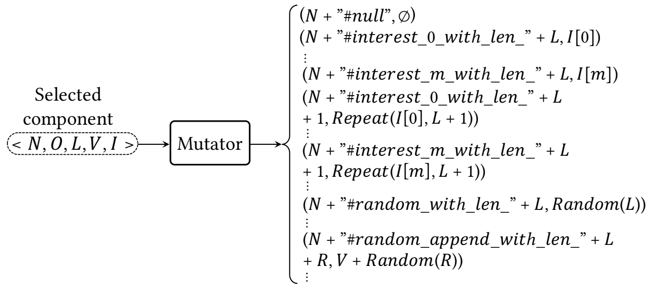
Selected component
$< N, O, L, V, I >$
Mutator
$(N + \text{"}\#null\text{"}, \emptyset)$
$(N + \text{"}\#interest\_0\_with\_len\_\text{"} + L, I[0])$
$(N + \text{"}\#interest\_m\_with\_len\_\text{"} + L, I[m])$
$(N + \text{"}\#interest\_0\_with\_len\_\text{"} + L + 1, Repeat(I[0], L + 1))$
$(N + \text{"}\#interest\_m\_with\_len\_\text{"} + L + 1, Repeat(I[m], L + 1))$
$(N + \text{"}\#random\_with\_len\_\text{"} + L, Random(L))$
$(N + \text{"}\#random\_append\_with\_len\_\text{"} + L + R, V + Random(R))$

Figure 4: Mutator inside the testcase generator

used by the combinatorial testing. In fact, our adaptive combinatorial test case generator uses the component specifications to construct the test cases; discussed below.

5.4 Combinatorial Testing

Now we discuss first how components are specified in combinatorial testing and then how one uses them to generate test-cases.

Component specification. A component is specified using 5-tuple $\langle N, O, L, V, I\rangle$, where N is of string type representing the component name; O is an integer type representing the order of the component within the whole sequence of byte string; L represents the correct length of the component (in bytes); V is the correct value for this component; and I is an array list of values of interest for the component. Note that, for a given component c, its field I is manually determined, possibly based on neighboring components. As an example, $\langle$"leading_byte", 0, 1, `0x00`, [`0x01`, `0xFF`]$\rangle$ describes leading byte component which mandates the test case generator to place its candidate value at the first of the generated sequence for the test case (*i.e.*, 0 index); and occupies 1 byte (*i.e.*, the correct length is 1) whose correct value is `0x00`. It also suggests some values of interest from [`0x01`, `0xFF`] list to be used in test case generation process. These specific values are suggested because these are also the values of some neighboring components.

Other parameters. Besides the above 17 components of the encoded message structure (EM_v), the signature verification algorithm accepts other arguments (e.g., size of modulus, hash algorithm, the received message). To reduce the search space, the test generation approach fixes those other parameters.

Component mutator. When the test generation approach selects a component c for analysis, the component mutator randomly chooses a set of values from the interesting values I as part of the c's component specification (See Figure 4). Each element of the interesting values I for a given component c is pair of the form $\langle \ell, b\rangle$ where the b field is actual byte sequence (in hexadecimal) to be used for testing whereas the ℓ field is a descriptive label used for debugging purposes capturing the exact mutation performed on the desired byte sequence. Note that, these interesting values can be described to be parametric and we use the following macros to define them. The $Repeat(x, y)$ macro returns a byte sequence that repeats y times the input byte sequence x whereas the $Random(y)$ macro returns a y-byte random byte sequence.

Test case generator. The combinatorial test case generator takes three arguments min, max, and C where C is the component specification (discussed before). It is responsible for generating test cases by mutating x components where $min \leq x \leq max$. At each step, it randomly selects x components to mutate. For each of those components, their corresponding component mutators are called. Each component mutator returns a random number of the mutated byte sequences for that component. A cartesian product is taken between the different component byte sequences where unmutated components use their original value. As an example, suppose $x = 2$ and hence two components c_i and c_j will be randomly selected for mutation. Let us also suppose that component mutators for c_i and c_j returned y and z, respectively, number of mutated values to consider. In that case, the test case generator will end up with $y \times z$ possible test cases. Note that, from these test cases, we throw away any test case which results in an encoded message whose length is greater than the RSA modulus as implementations trivially reject those test cases.

5.5 Adaptive Test Generation

At each step, our adaptive domain-specific test generation strategy randomly selects a component c_1. It then tries to randomly select a byte to steal from c_1 and hide it in all other remaining components c_2. This results in c_1 to decrease its length by 1 whereas c_2's length increases by 1. Stealing byte is analogous to deleting a byte from c_1 whereas hiding a byte in c_2 is analogous to adding a random byte to c_2. If the stealing and hiding operations result in a signature rejected by the oracle but accepted by the implementation-under-test, then the test generator would try to steal and hide more bytes, and in some cases also search the range of accepted values. This way, the test generator can identify the amount of bytes (and values) the attacker can use to launch a Bleichenbacher-style low public exponent RSA signature forgery attack.

6 EVALUATION

In this section, we first demonstrate the efficacy of Morpheus in finding non-compliance in recent implementations, and then compare it against some general-purpose fuzzers and a recent work [33].

6.1 Findings

We evaluated Morpheus against 45 implementations of PKCS#1-v1.5 signature verification and found 9 of them are lenient, while 33 implementations have incompatibility issue. Table 2 shows a summary of Morpheus's test results against various PKCS#1-v1.5 implementations. Among 9 lenient implementations, we have discovered 6 of them suffer from Bleichenbacher-style low public exponent RSA signature forgery, 1 implementation is susceptible to buffer overflow attack, and 8 implementations with some minor leniencies (*i.e.*, accepting signatures that should have been rejected but the leniencies are such that they cannot be exploited for signature forgery). There are also 5 implementations with no bugs found. Here we discuss the significant findings and the related attacks, and leave the details of other leniency in Appendix D. For details on the concrete parameter values used in our evaluation, see Appendix B.

6.1.1 *node-forge (v0.10.0).*

Forge library [3] contains a native implementation of the TLS protocol in JavaScript and a set of cryptography utilities for application development. The PKCS#1-v1.5 signature verification in node-forge employs decoding approach. Using Morpheus, we found that it suffers from the following exploitable vulnerabilities in its verification logics.

1) Accepting less than 8 bytes of padding: The node-forge PKCS#1-v1.5 signature verification implementation does not check whether PS has a minimum length of 8 bytes. After root-cause analysis, we found that after the block type value is checked to be `0x01` (line 1 in Appendix D.1.1), the implementation skips all padding bytes until it reaches to the end of padding. This leniency enables an attacker to steal all the padding bytes and then inject new bytes (with same length as those stolen) in other places that are left unchecked, which can be used together with the other findings below for signature forgeries.

2) Ignoring digestAlgorithm structure (CVE-2021-30247): Once the encoded message is obtained from modular exponentiation, node-forge decodes the structured message to obtain the hash value. However, we found that node-forge only checks the obtained `digest` value (by comparing it against the computed hash value of the received message) and ignores verifying the decoded `digestAlgorithm` structure, leaving some unchecked area exploitable for attacker to launch signature forgery.

3) Accepting trailing bytes (CVE-2021-30249): node-forge fails to check that after `DigestInfo` ASN.1 structure, there should not be any trailing bytes. This is a classical flaw previously shown in other implementations as well [38]. Together with the fact that node-forge accepts less than 8 bytes of padding as described above, this creates an large unchecked room exploitable for signature forgery.

◦ *Signature forgery:* Based on our findings, node-forge would accept a malformed $\widehat{EM}_v$ in the form of `0x00 || 0x01 || 0x00 || T || GARBAGE`. Knowing this, the attacker can prepare $\widehat{EM}_v$ where `T` contains a hash of an attacker-chosen message $\widehat{m}$, and `GARBAGE` contains some fixed values (*e.g.*, all `0xFF..FF`) as the trailing bytes. One can then take the `e`-th root of $\widehat{EM}_v$ to find the attack signature $\widehat{S}$ (without knowing the private exponent). Notice that the attacker-prepared $\widehat{EM}_v$ might not be a perfect power of `e`, and thus $\widehat{S}$ might not be the perfect `e`-th root of $\widehat{EM}_v$. However, as long as the imprecision stays in the unchecked trailing `GARBAGE`, the signature forgery would succeed. To further maximize the number of unchecked `GARBAGE` bytes, we can exploit the vulnerability of ignoring `digestAlgorithm` to modify `T` such that the `digestAlgorithm` structure in `DigestInfo` encoded structure is replaced with a minimum number of bytes that makes the decoding operation of node-forge pass through (e.g., `0x0100` where `0x01` is the ASN.1 tag for boolean and `0x00` denotes a zero-length content). This works because after the value is decoded, it is not going to be checked by the verifier, and shortening the `digestAlgorithm` structure allows the attacker to have even more trailing garbage bytes.

The difficulty of finding a working $\widehat{S}$ is bound by the the distance between two consecutive perfect power of `e`. For instance, when

`|n|` = 2048 bits and assuming `SHA-256` hash function being used, we end up having 1720 bits for trailing garbage bytes (*i.e.*, 2048 − 24 (3 prefix bytes) − 304 (for encoding `T'` with `SHA-256`) = 1720). Given `e = 3` and letting k^e be the integer representation of $\widehat{EM}_v$, we can upperbound the the attacker-prepared encoded message by $k^3 < 2^{(2048-15)} = 2^{2033}$. So, the distance between two consecutive perfect cubes in this range becomes:

$$k^3 - (k-1)^3 = 3k^2 - 3k + 1 < 3 \cdot 2^{1356} - 3 \cdot 2^{678} + 1 < 2^{1358}$$

which is less than the unchecked trailing bytes (*i.e.*, $< 2^{1720}$), and thus the forgery attempt will always work (the imprecision will always stay in `GARBAGE` and be left unchecked). This can be seen as an improved variant of Bleichenbacher's original attack [38, 52].

In fact, because of these bugs in node-forge, signature forgery will succeed even with a slightly larger e. Under the same set of security parameters but $e = 5$, we can upperbound the attacker-prepared encoded message by $k^5 < 2^{(2048-15)} = 2^{2033}$, and the distance between two consecutive perfect 5-th power becomes $k^5 - (k-1)^5 < 2^{1630}$, which is again less than the unchecked trailing bytes (*i.e.*, $< 2^{1720}$). Similarly, when `|n|` = 8192 bits and $e = 17$, we have 7862 bits of trailing `GARBAGE` left unchecked if `SHA-256` hash is used. The attacker-prepared encoded message can then be upperbound by $k^{17} < 2^{(8192-15)} = 2^{8177}$, and the distance between two consecutive perfect 17-th power can be bound as $k^{17} - (k-1)^{17} < 2^{7701}$, which is again less than the unchecked trailing bytes (*i.e.*, $< 2^{7862}$). This shows that with such bugs in the implementation, the use of a longer RSA modulus actually further weakens instead of improves security.

6.1.2 *wpa_supplicant & hostapd (v2.9).*

Both wpa_supplicant [8] and hostapd [4] can be configured to use OpenSSL, GnuTLS, or their own internal TLS implementation (marked as experimental by the maintainer). A quick inspection shows that wpa_supplicant and hostapd actually share the same internal TLS implementation, which is unsurprising given the two are maintained by the same developer. Since we will revisit new versions of OpenSSL and GnuTLS next (and they have been thoroughly tested in previous works [33, 52]), here we focus on testing the PKCS#1-v1.5 signature verification in their internal TLS implementation.

1) Lax length octet checking for DER: Both implementations suffer from the lack of proper checking of length octet while decoding ASN.1 structure of EM_v with respect to DER [10]. In DER, definite long form encoding of a length less than 128 bytes are not allowed. For instance, a length of 9 should be encoded as $0x09$, but these implementations incorrectly allow the definite long form encoding, such as $0x8109$, where the first byte (with the MSB set) shows the number of subsequent bytes need to be concatenated to obtain the length value. This mistake not only makes the implementations overly lenient in accepting incorrect signature values (whose EM_vs contain incorrect encoding of length octets) but also contribute to the signature vulnerability discussed below. In the benign cases under the commonly supported hash functions, it is not necessary for implementations to support the definite long form encoding of length octets (*i.e.*, there are no content octets with length larger than 127). This alone does not lead to an immediate attack, however, attackers can maximize the size of unchecked bytes when exploiting other bugs as discussed below.

2) Leniency in checking `AlgorithmIdentifier` structure (CVE-2021-30004): Using Morpheus, we found that both wpa_supplicant and hostapd have the classic flaw of not checking the algorithm parameter component. Once OID (*i.e.*, object identifier) of the hash function is decoded, the implementations skip the rest of the `AlgorithmIdentifier` structure, which contains the parameter field for the hash function being used. This is similar to the bug in strongSwan 5.6.3 reported by a recent work [33], as well as in Mozilla NSS [31] and GnuTLS [43] many years ago. This can be exploited for signature forgery, and the resulting unchecked area can be expanded by the definite long form of length octets discussed above.

∘ *Signature forgery:* Given the above vulnerability, these implementations would accept a malformed $\widehat{EM}_v$ in the form of

```
0x00 || 0x01 || PS || 0x00 || A || GARBAGE || B
```

where `PS` is `0xFFFFFFFFFFFFFFFF` as these implementations require a minimum of 8 bytes of padding; `A` is the octet string containing the initial portion of ASN.1 encoded structure (*i.e.*, `T`) up to the hash algorithm parameter field; `GARBAGE` indicates the bytes in `T` corresponding to the unchecked hash algorithm parameter field; and finally, `B` is the octet string containing the last portion of `T` representing the TLV for hash value (*i.e.*, the `digest` structure). More concretely, when `|n| = 2048`, `H = SHA-256`, and to-be-forge message `m' = "hello world!"`, then:

```
A  = 0x3081f13081cd060960864801650304020l
B  = 0x04207509e5bda0c762d2bac7f90d758b5b2263fa01ccbc542ab5e3df16
     3be08e6ca9
```

The length octets in `A` have been adjusted to use definite long form whenever possible (*e.g.*, the lenght of `0x81f1` and `0x81cd` for the `DigestInfo` and `AlgorithmIdentifier` structures, respectively) to maximize the length of `GARBAGE`.

Since the unchecked area resides in the middle of $\widehat{EM}_v$, an attacker can forge a signature $\widehat{S} = (k_1 + k_2)$, where k_1 and k_2 are two integers chosen such that once $\widehat{S}$ is processed by the verifier (*i.e.*, when `e = 3`, verifier obtains $\widehat{EM}_v = \widehat{S}^3 = (k_1 + k_2)^3 = k_1^3 + 3k_1^2 k_2 + 3k_2^2 k_1 + k_2^3$ after processing $\widehat{S}$), the following properties hold for the malformed encoded message $\widehat{EM}_v$:

(1) The most significant bits of k_1^3 should match `0x00 || 0x01 || PS || 0x00 || A`, the octet string right before the unchecked area;
(2) The least significant bits of k_2^3 should match `B`, the octet string right after the unchecked area;
(3) The least significant bits of k_1^3 and the most significant bits of k_2^3 along with $3k_1^2 k_2 + 3k_2^2 k_1$, should stay in the unchecked area, indicated by `GARBAGE`.

Success of the forgery attempt hinges on whether there are enough unchecked bytes to be exploited. Otherwise, the terms of $(k_1 + k_2)^3$ expansion would overlap with each other, and as a result, it becomes difficult for the above properties to hold. However, according to the above vulnerabilities, the number of unchecked bytes grows linearly with larger `|n|` (fixing the same public exponent), which is yet another example of longer RSA modulus further weakens instead of strengthens security.

How to find k_1: Finding k_1 is quite similar to finding the attack signature for the case of trailing garbage bytes, as we have seen in attacking node-forge. That is, constructing `C = 0x00 || 0x01`

`|| PS || 0x00 || A || GARBAGE` with the length equals to `|n|` and then find the cubic root. However, here we cannot hide the inaccuracy of cubic root of the malformed construct in the least significant bits of the obtained encoded message. Therefore, we need to adjust the k_1 such that once raised to the power `e = 3`, the obtained encoded message contains as many as zero bits in its least significant bits. For that, we compute $t_1 = \lceil \sqrt[3]{C} \rceil$ and then sequentially search for the largest possible r such that $((t_1/2^r + 1) \cdot 2^r)^3$ yields a number of the form `0x00 || 0x01 || PS || 0x00 || A || GARBAGE`. Once the largest r has been found, then we can compute $k_1 = (t_1/2^r + 1) \cdot 2^r$.
How to find k_2**:** Finding k_2 is similar to finding signature attack for the case we have forwarding garbage bytes in an encoded message, as in `GARBAGE || B`. In this case the the modular exponentiation can be seen as computed over a much smaller modulus n', instead of n. In RSA, finding $\phi(n)$ often requires factorizing n, which is believed to be impractical when n is large, now by having a special n' it is easy to compute $\phi(n')$. The attacker first needs to compute $n' = 2^b$ where $b = |\mathtt{B}|$, and then compute $\phi(n') = \phi(2^b) = 2^{b-1}$, because $\phi(p^i) = p^i(1 - 1/p)$ when p is prime and $i \geq 1$. Notice that k_2 should satisfy $k_2^e \equiv B \pmod{n'}$. Since n' is a power of 2, we can guarantee k_2 and n' are coprime by choosing an odd numbered `B` with a fitting hash value. By knowing $\phi(n')$ and $e = 3$, attacker can find a fake private exponent f which acts similar to d. Attacker uses Extended Euclidean Algorithm to find f such that $ef \equiv 1 \bmod \phi(n')$ and use f to compute k_2 as $k_2 = B^f \bmod n'$.

∘ *Real-world impact:* Because of this flaw, systems (*e.g.* resource-constrained embedded devices) that use v2.9 of wpa_supplicant or hostapd and rely on its internal TLS implementation, will be susceptible to signature (certificate) forgery attacks. This can be exploited in tandem with a WPA2-Enterprise Evil Twin attack for stealing user credentials [24], especially when the WPA2-Enterprise setup is configured to use the system CA store as the trust anchor for certificate validation.

Despite the experimental nature of the internal TLS implementation, we found that the signature forgery discussed above can indeed lead to practical attacks in the real world. We purchased a commodity Wi-Fi router and replaced its factory firmware with OpenWRT, one of the most popular open-source Linux-based router firmware, and installed the `ca-bundle`, `wpa-supplicant` and `hostapd` packages from its package manager (`opkg`). The OpenWRT `ca-bundle` package is based on the same set of trusted CA certificates distributed and used by Debian Linux, and contains two CA certificates that have a public key with $e = 3$, namely the *Go Daddy Class 2 Certification Authority* and the *Starfield Class 2 Certification Authority*. In fact, as long as there is one such certificate included in the trust anchors, a certificate forgery attack exploiting the signature verification flaw can succeed. We generated an attack certificate pretending to be issued by Go Daddy (by filling in appropriate information in the `issuer` field), and followed the aforementioned steps to forge a signature that can trick wpa_supplicant. Then we used a Raspberry Pi 4 to act as the Evil Twin and tried to trick the wpa_supplicant running on OpenWRT (configured to establish a WPA2-Enterprise EAP-TTLS Wi-Fi connection automatically, using the system CA store to validate the certificate of the authentication server). With the help of the attack certificate, the Evil Twin was able to pass the certificate validation of wpa_supplicant, pretend to be the legitimate authentication server, and steal user credentials. The same attack also works if the victim supplicant setup is configured to use PEAP instead of EAP-TTLS.

Similarly, for hostapd, under the WPA2-Enterprise EAP-TLS mode, users typically perform a certificate-based authentication to the server during TLS handshake. However, due to the signature verification flaw, we were able to generate fake certificates that appear to be issued by a legitimate authority (*e.g.*, Go Daddy), and then gain access to a EAP-TLS WPA2-Enterprise Wi-Fi network safeguarded by hostapd on OpenWRT as any legitimate users.

6.1.3 *RELIC (git commit 32eb4c25).*

RELIC [7] is a cryptographic library developed with the goal of improving efficiency and flexibility. Therefore, it can be tailored to meet specific security levels and algorithmic choices. RELIC's PKCS#1-v1.5 signature verification uses decoding approach to parse the given signature's encoded message, however, as identified by MORPHEUS, it is susceptible to signature forgery and buffer overflow attacks.

1) Leniency in checking the prefix bytes (CVE-2020-36315): The implementation's signature verification logic is lenient in checking the prefix bytes of EM_v, which includes leading byte, block type byte, and padding strings. For the first two bytes, although there are some checks in place to identify the errors, the implementation mistakenly continues the verification operation (instead of terminating it) when the errors occur, and as a result, the identified errors are eventually overwritten. Afterwards, the code attempt to peel of the padding bytes without checking their values, until the end-of-padding zero is reached. Because of this, a new variant of signature forgery attack is possible.

∘ *Signature forgery:* We can exploit this with a similar strategy used to forger signatures for wpa_supplicant and hostapd. The only difference is that the unchecked (padding) bytes cannot contain the value of zero. This affects how we find k_2. The trick is that, after finding k_1 and k_2 like before, we set some randomly chosen high bits of k_2 to 1, so that the output of $k_2^3 \bmod n$ might have random but non-zero padding. Assuming we can sample output values of the modular exponentiation uniformly at random by this trick, the success probability is $(\frac{255}{256})^x$ where x is the size of the padding bytes, which amounts to about 40% chance of success under $|n| = 2048$ bits and SHA-256 hash. To the best of our knowledge, this is a new attack variant not discussed by previous work.

2) Buffer overflow caused by missing length checks (CVE-2020-36315): Another vulnerability identified by MORPHEUS, is caused by the lack of necessary checks to ensure that components of EM_v have correct lengths. After peeling off the prefix bytes, the implementation checks T's ASN.1 encoded content up to hash value bytes. Afterwards, it copies all bytes of the hash value to a pre-allocated buffer using a *computed length* based on the size of padding it peeled off. Because RELIC also does not enforce proper length checks on the padding, it is possible for an attacker to use a very short padding in an attack signature to mislead the code into using a very large *computed length*, larger than the size of the pre-allocated buffer, when attempting to copy the hash value, hence inducing a buffer overflow. Such attack signatures can be generated similar to the signature forgery discussed for node-forge. As a proof of concept, we managed to forge signatures that can crash a verifier using RELIC.

6.1.4 phpseclib v3.0 (relaxed mode). phpseclib [6] is a PHP library that provides pure-PHP implementations of cryptographic protocols with active supports for three versions (v1.0, v2.0, v3.0). phpseclib v3.0 has two modes of operation for PKCS#1-v1.5 signature verification, known as relaxed and strict modes. The relaxed mode of phpseclib v3.0 is implemented to support BER encoding of ASN.1 data value in the encoded message obtained from the signature value. The differences between DER and BER occur mainly in length octets: DER does not allow indefinite length form and forbids the definite long form when the length can be encoded directly. Interestingly, phpseclib v3.0 (relaxed mode) uses the decoding approach for signature verification, while the other versions all take the encoding-based path. Unfortunately due to leniency in its parser, phpseclib v3.0 (relaxed mode) suffer from 2 vulnerabilities which enable signature forgery, and 2 bugs causing interoperability issue.

1) Leniency in checking parameter field (CVE-2021-30130) Similar to the second bug of wpa_supplicant and hostapd, phpseclib v3.0 (relaxed mode) fails to properly check the parameter field associated to the hash algorithm in `AlgorithmIdentifier` structure. Once it parses the NULL tag (*i.e.*, `0x05`), it returns an empty string without actually checking the length is zero and there is no content octets to consume. In fact, the tag does not even have to be NULL because any tags will be accepted by the parser. This bug leaves an unchecked area enabling an attacker to launch signature. Given that indefinite form for length octet is allowed in BER and supported by phpseclib v3.0 (relaxed mode), attacker can further maximize this unchecked area to increase the chance of success. Notice that the signature forgery attack is similar to that of wpa_supplicant and hostapd, where we have `GARBAGE` bytes in the middle of the malformed encoded message.

2) Leniency in decoding hash function OID's content octets The `AlgorithmIdentifier` structure contains a TLV encoding of the hash function OID used during the sign operation on `m`. OIDs (object identifiers) are identifier mechanism standard [9] for naming any object with a globally unambiguous persistent name, where it has its own encoding/decoding rules. The implementation, however, has a logical flaw in decoding a dotted decimal integer whose value is is greater than or equal to 128. Therefore, an attacker can inject garbage bytes after a correct hash algorithm OID the signature still verifies. The injected garbage bytes cannot be arbitrarily chosen, where every byte has to have its most significant bit set (*i.e.*, its value should be in this range [`0x80`, `0xff`]). That being said, it is not practical to launch Bleichenbacher-style low public exponent RSA signature forgery because the garbage bytes injected in the middle cannot be freely chosen; however, it has interoperability issue where an invalid signature is mistakenly accepted.

6.1.5 Incompatibility issues. The PKCS#1-v1.5 specification has evolved since it was first proposed, and historically there were some confusions over whether the presence of a NULL parameter field of `AlgorithmIdentifier` is optional, sometimes leading to incompatibility issues among different implementations. According to the latest revision of the standard as described in RFC8017, *"for the SHA algorithms, implementations MUST accept `AlgorithmIdentifier` values both without parameters and with NULL parameters"*. However, we found that there are significant number of implementations (33 of them in Table 2) that reject the implicit NULL parameter case. Although most RSA certificates have the explicit NULL parameter field, our empirical study on Censys [2] certificates dataset found that the implicit NULL parameter case still exists on 4% of RSA certificates, which suggests this issue can indeed damage interoperability. We detail the evolution of the specification in Appendix A.

6.2 Comparison with different approaches

6.2.1 General-purpose fuzzers. We now empirically show the rationale for designing a new input sampler instead of using the mutation engine of an existing fuzzer. The main reasons that contribute to the complexity of generating effective test cases for analyzing PKCS#1-v1.5 signature verifiers are as follows. First, we aim at covering many implementations (written in different languages, for different platforms/architectures, and sometimes proprietary), thus we consider the most generic black-box setting without instrumentation. This limits the use of rich source-level feedback (e.g., coverage metrics) for generating effective test cases. Second, even if one assumes a 512-bit modulus (which is not recommended by today's security standard [48]), a fuzzer with random test case generation will have to deal with EM_v inputs of 512-bit long, effectively exploring in a space of 2^{512} test cases. Combining with the first reason, this substantially reduces the chance of exposing interesting non-compliance. Third, a PKCS#1-v1.5 encoded message is highly structured and requires a context-sensitive grammar to faithfully capture the format. Bit-level mutation strategy is unlikely to be successful in exposing interesting non-compliances.

Fuzzer selection. We selected AFL [13] and AFL++ [14] to compare against Morpheus. As we consider a purely black-box setting, our evaluation is thus a comparison between mutation strategies without source-level feedback. Roughly, most fuzzers differ from each other in the type of feedback (e.g., code coverage) and how they use the feedback. We chose AFL because of its generality and effectiveness in identifying weaknesses in a wide-variety of applications. On the other hand, as the PKCS#1-v1.5 encoded message is highly structured one can envision that a representative (context-free) grammar-based fuzzer like AFL++ may have a better chance in generating interesting test cases.

Incorporating AFL and AFL++ into Morpheus workflow. Like many other general-purpose fuzzers, AFL and AFL++ are designed to discover inputs that can cause a crash in the target program. We replaced our input sampler with AFL and AFL++ in the Morpheus architecture (See Figure 1). We have, however, observed that by just running AFL and AFL++ against those subject implementations do not lead to any interesting discovery as the semantic bugs reported by Morpheus do not necessarily cause a crash in the system. Hence, we tested the bug detector component by modifying it to generate an artificial crash whenever a deviation is found and thus provide AFL/AFL++ with some behavioral feedback.

Configuring AFL and AFL++. AFL takes seed inputs and applies a variety of mutation techniques to generate new test cases. Seeds used to run AFL (and, AFL++) contain correct PKCS#1-v1.5 signature structures for some given fixed parameters (e.g., hash algorithm, modulus size), similar to what is presented in Appendix section B. Although the mutator of AFL is grammar-blind, we take advantage of the support of user-defined dictionaries to make it aware of the basic structure of the test cases in order to maximize its effectiveness. We created our own dictionary containing meaningful/interesting byte sequences that may occur in PKCS#1-v1.5 signature structure irrespective of the message being signed.

AFL++ is based on AFL and supports custom mutators. We use the open-source grammar mutator [17] and configured AFL++ to only use this mutator to evaluate the grammar mutator performance. We then created a context-free grammar — *an approximation to describe the PKCS#1-v1.5 structure* — in order to give it some guidance on dealing with the input structure.

Results. Our evaluation on AFL and AFL++ adopts suggestions from Klees et al. [49] and refers to the setups in the latest fuzzers UNIFUZZ [54] and WINNIE [44]. We launched 5 trials for each target implementation and ran 24 hours for each trial. We omitted Apple Security Framework's Crypto and STM32-crypto for this set of comparison experiments since they cannot be run natively on a x86-64 Ubuntu 20.04 machine, where we have the setups of AFL and AFL++. As can be seen in Table 2, AFL and AFL++ are able to detect some of the bugs reported by Morpheus, with the help of our oracle as well as domain knowledge. However, their test case generations still struggle to generate high quality test cases to reveal all the bugs discovered by Morpheus, and they also did not detect any new bugs not caught by Morpheus. It is also interesting that when AFL++ was configured to only use the grammar mutator, it did not outperform AFL's basic mutators in our case study.

6.2.2 *A KLEE-based approach in previous work [33].*

We first use Morpheus to revisit the old implementations that were found to be problematic in [33]. The test results can be found in Table C1 in Appendix C. We note that all of the flaws that can lead to signature forgery and buffer overflow as reported by previous work can also be found by Morpheus. Moreover, we realized that due to how symbolic execution was being setup in previous work, Morpheus actually uncovered more findings regarding incompatibility and minor leniency. First, previous work did not investigate the incompatibility issues regarding an implicit NULL algorithm parameter. More intricately, because of scalability reasons, previous work used concrete bytes for ASN.1 tags [33] in all its test harnesses, and thus cannot detect leniency in the parsing of tags (*e.g.,* ignoring the class and form bits). Similarly, with respect to the 3 test harnesses (TH1, TH2, TH3) used by previous work, the ASN.1 length variables take concrete values in TH1 and TH2, and the symbolic length variables used by TH3 is always 1-byte long [33], and thus KLEE was not given enough symbolic bytes to explore potential leniency related to long and indefinite forms of encoding ASN.1 length.

Finally, to give a comprehensive comparison, we also applied the KLEE toolchain used by Chau *et al.* [33] on the few applicable new subjects that we tested with Morpheus. The relevant statistics can be found in Table C2 in Appendix C. Among the open source ones covered in Table 2, only IPP Crypto, wpa_supplicant, Apache milagro, and RELIC were written in `C`. However, IPP Crypto requires `clang` version 9 or above (whereas the toolchain uses version 3.4), thus we exclude it in the comparison. For making wpa_supplicant and RELIC amenable to the approached used in [33], 11 and 8 lines in the source tree were modified respectively, for injecting the concolic test buffer. Apache milagro did not require any source tree modifications due to its API design.

Table 2: Bugs found by Morpheus, AFL, and AFL++

Implementation name	Source code language	Bug[1]	Morpheus	AFL	AFL++[2]
wpa_supplicant v2.9	C	SF#1	✓	✓	X
		ML#1	✓	X	X
		ML#2	✓	✓	✓
		ML#3	✓	X	X
		ML#4	✓	✓	✓
hostapd v2.9	C	SF#1	✓	✓	X
		ML#1	✓	X	X
		ML#2	✓	✓	✓
		ML#3	✓	X	X
		ML#4	✓	✓	✓
IPP Crypto v2020 Update 3	C	IN	✓	✓	✓
SunRsaSign OpenJDK v11.0.10	Java	-	-	-	-
Amazon Corretto Crypto Provider v1.5.0	Java	IN	✓	✓	✓
Bouncy Castle Provider v1.67	Java	-	-	-	-
Rust Crypto RSA v0.3.0	Rust	IN	✓	✓	✓
pyca/cryptography v2.1.4	Python	IN	✓	✓	✓
phpseclib v1.0	PHP	IN	✓	✓	✓
phpseclib v2.0	PHP	IN	✓	✓	✓
phpseclib v3.0 (relaxed mode)	PHP	SF#1	✓	✓	X
		ML#1	✓	✓	✓
		ML#2	✓	✓	✓
		ML#3	✓	X	X
phpseclib v3.0 (strict mode)	PHP	IN	✓	✓	✓
CryptX v0.070	Perl	ML#1	✓	✓	✓
cryptonite v0.28	Haskell	IN	✓	✓	✓
PyCryptodome v3.10.1	Python	-	-	-	-
buddy-core v1.9.0	Clojure	-	-	-	-
jsrsasign v10.1.13	JavaScript	ML#1	✓	✓	X
		IN	✓	✓	✓
node-forge v0.10.0	JavaScript	SF#1	✓	✓	✓
		SF#2	✓	X	X
		ML#1	✓	X	X
		ML#2	✓	✓	✓
Node Crypto v14.16.0	JavaScript	IN	✓	✓	✓
Node-RSA v1.1.1	JavaScript	IN	✓	✓	✓
node-jws v4.0.0	TypeScript	IN	✓	✓	✓
Apple Security Framework's Crypto v2021.03.26	Swift	IN	✓	N/A	N/A
asmCrypto v2.3.2	JavaScript	IN	✓	✓	✓
jose-jwt v0.9.1	Haskell	IN	✓	✓	✓
go rsa v1.16.2	Go	IN	✓	✓	✓
Solidity RSA PKCS1 Verification git commit b927ddb	Solidity	IN	✓	✓	✓
Apache milagro v2.0.1	C	IN	✓	✓	✓
py-pkcs1 v0.9.6	Python	IN	✓	✓	✓
Crypto++ v8.5	C++	IN	✓	✓	✓
RELIC git commit 32eb4c25	C	SF#1	✓	✓	X
		BO#1	✓	X	X
		IN	✓	✓	✓
Seed7 git commit 1e4e942 [3]	Seed7	SF#1	✓	✓	X
		SF#2	✓	✓	X
		SF#3	✓	✓	✓
		SF#4	✓	X	X
		ML#1	✓	✓	✓
		ML#2	✓	✓	✓
Microsoft .Net Cryptography v5.0	C#	IN	✓	✓	✓
GaloisInc RSA v2.4.1	Haskell	IN	✓	✓	✓
STM32-crypto v3.1.0	Firmware binary	IN	✓	N/A	N/A
axTLS v2.1.5	C	IN	✓	✓	✓
MatrixSSL v4.3.0	C	-	-	-	-
MbedTLS v2.26.0	C	IN	✓	✓	✓
LibTomCrypt v1.18.2	C	ML#1	✓	✓	✓
strongSwan v5.9.2	C	IN	✓	✓	✓
Openswan v3.0.0	C	IN	✓	✓	✓
Erlang's public_key v11.1.7	Erlang	IN	✓	✓	✓
tsec v0.2.1	Scala	IN	✓	✓	✓
OpenSSL v1.1.1k	C	IN	✓	✓	✓
OpenSSL v3.0.0-alpha15	C	IN	✓	✓	✓
GnuTLS v3.6.15	C	IN	✓	✓	✓

[1] SF: Signature Forgery; ML: Minor Leniency; BO: Buffer Overflow; IN: INcompatibility issue
[2] Configured to only use the grammar mutator
[3] At the time of writing, we are not aware of applications that use its TLS implementation

In the end, similar to Morpheus, no unwarranted leniency was found in Apache milagro. For RELIC, the KLEE-based approach used in [33] also found its leniency in checking the prefix bytes [SF#1], as well as its acceptance of trailing bytes after the hash digest. However, because of how it was setup to avoid scalability issues, only 2 bytes can be moved around [33], and thus KLEE did not directly uncover the potential buffer overflow [BO#1]. For wpa_supplicant, the KLEE-based approach also found its incorrect extended tag decoding of identifier octets [ML#3], leniency in checking the length octet of `DigestInfo` [ML#4], and leniency in checking `AlgorithmIdentifier` structure [SF#1]. However, this approach was unable to find the lax length octet checking for DER [ML#1] and leniency in checking form bit of an identifier octet [ML#2] due to limitations of its test harnesses as discussed above.

7 RELATED WORK

PKCS#1-v1.5. Despite its theoretical security guarantees [40], many implementations of PKCS#1-v1.5 signature scheme exhibit flaws that can lead to signature forgery, which were identified through manual inspection and hand-crafted test cases [1, 11, 12, 30, 38, 42, 53]. A recent work proposed an approach based on symbolic execution [33], but many implementations were not considered due to limitations of the toolchain, which greatly motivates this work. Previous work also attempted to implement a secure parser for the ASN.1 portion of PKCS#1-v1.5 signatures [58], but to the best of our knowledge it does not directly enforce all the signature verification requirements stipulated by the specification.

Orthogonal to its signature scheme, the PKCS#1-v1.5 cryptographic standard also includes an encryption scheme, and many implementations of which were found to be susceptible to a padding oracle attack also attributed to Daniel Bleichenbacher [26]. This padding oracle is notoriously difficult to avoid, as the leakage can manifest via various side channels, including but not limited to error messages [27, 50] and timing [41, 55, 59].

Finding logical bugs. Fuzzers (such as AFL [13], LibFuzzer [19], and Honggfuzz [18]) have shown to be effective in finding low-level memory bugs and runtime errors. Scalable infrastructures (*e.g.*, OSS-Fuzz [21], ClusterFuzz [15], and FuzzBench [16]) have been designed to promote their use. Several fuzzers (*e.g.*, AFL++ [14], SPIKE [23], and Peach [22]) support some forms of grammar representation, mostly based on a context-free grammar, to allow them to explore deeper in the source code when dealing with structured inputs. However, whether these techniques are effective in discovering logical bugs often depends on the problem at hand. One potential issue is that context-free grammar might not be sufficiently expressive to capture the intricacies of the input format, possibly generating many test cases that are trivially rejected. Another general challenge concerns the lack of an oracle which prior work alleviates by adopting differential testing [29, 32, 33, 57].

8 DISCUSSION

Formalization. We manually go over the English specification of the PKCS#1-v1.5 standard and formalized it in Gallina. It is thus our interpretation of the specification. Unfortunately, there are currently no approaches to verify that the correctness requirements we formalized indeed correspond to the natural language description.

Differential testing. A differential testing approach alleviates the lack of a test oracle by comparing pairs of implementations-under-test to find behavioral deviations. The implementations are used as cross-checking oracles. When a deviation is detected, it is not clear which implementation is noncompliant, requiring manual intervention to triage the noncompliance. Also, when the compared implementations suffer from the same noncompliance, then it will get undetected. Both of these weaknesses are subverted by Morpheus by using a formally proven test oracle.

Limitations. Morpheus's oracle currently recognizes the object identifiers of the SHA family hash algorithms. Signatures using other hash algorithms will thus be wrongly rejected by it. In addition, the correctness of PKCS#1-v1.5 oracle critically hinges of the correctness of Coq's extraction mechanism as well as the OCaml's toolchain. The PKCS#1-v1.5 oracle does not perform the cryptographic operations for a signature verification and instead only focus on the format checking of the PKCS#1-v1.5 standard. Finally, Morpheus being a testing approach is incomplete, that is, it is not guaranteed to discover all noncompliances of an implementation.

Manual efforts. Realizing Morpheus required manual efforts in the following three aspects: (1) formalizing the correctness theorem by consulting the standard; (2) developing the oracle and proving its correctness; (3) generate a test harness to test a given implementation, which sets up the key, signature and message, and invokes the corresponding signature verification procedure. Among these, (1) and (2) are one-time efforts whereas (3) has to be carried out for any new implementations to be tested. In our case, (1) and (2) required approximately 180 person-hours whereas depending on the availability of sample code and document, (3) required a couple of person-hours on average for each implementation-under-test.

9 CONCLUSION

We developed an automated black-box non-compliance checker dubbed Morpheus that we used to check compliance of 45 various PKCS#1-v1.5 signature verification implementations developed in 18 different programming languages. Our analysis revealed that 40 implementations are non-compliant and 6 out of them suffer from Bleichenbacher-style low public exponent RSA signature forgery. The results suggest that variants of the Bleichenbacher-style low public exponent RSA signature forgery attack still work even after its initial discovery more than a decade ago.

Acknowledgments

We thank the reviewers for their insightful comments and suggestions on how to improve this paper. We would also like to thank the developers for taking the time to investigate and fix the issues found by Morpheus. This work was supported in part by the departmental startup budget NEW/SYC, GRF matching fund GRF/20/SYC, and Project Impact Enhancement Fund 3133292C from The Chinese University of Hong Kong (CUHK), as well as US Department of Defense (DARPA) Grant D19AP00039, and US National Science Foundation (NSF) grants CNS-2007512 and CNS-2006556. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the policies or endorsements of the funding agencies.

REFERENCES

[1] [n.d.]. BERserk Attack – Intel Security web Archive. https://web.archive.org/web/20150112153121/http://www.intelsecurity.com/advanced-threat-research/. Accessed: Apr 04, 2021.

[2] [n.d.]. Censys.io - Attack Surface Scan - Total Visibility Internet-Wide. https://censys.io/certificates. Accessed: Apr 04, 2021.

[3] [n.d.]. Forge – A native implementation of TLS in Javascript and tools to write crypto-based and network-heavy webapps. https://github.com/digitalbazaar/forge. Accessed: Apr 04, 2021.

[4] [n.d.]. hostapd – IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator. https://w1.fi/hostapd/. Accessed: Apr 04, 2021.

[5] [n.d.]. ipsec_rsasigkey - generate RSA signature key. https://manpages.debian.org/testing/libreswan/ipsec_rsasigkey.8.en.html. Accessed: Apr 04, 2021.

[6] [n.d.]. phpseclib – PHP Secure Communications Library. https://github.com/phpseclib/phpseclib. Accessed: Apr 04, 2021.

[7] [n.d.]. relic – Modern cryptographic meta-toolkit with emphasis on efficiency and flexibility. https://github.com/relic-toolkit/relic. Accessed: Apr 04, 2021.

[8] [n.d.]. wpa_supplicant – Linux WPA/WPA2/IEEE 802.1X Supplicant. https://w1.fi/wpa_supplicant/. Accessed: Apr 04, 2021.

[9] [n.d.]. X.660 : Information technology - Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree. https://www.itu.int/rec/T-REC-X.660. Accessed: Apr 04, 2021.

[10] [n.d.]. X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). https://www.itu.int/rec/T-REC-X.690/. Accessed: Apr 04, 2021.

[11] 2006. CVE-2006-4340. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4340. Accessed: Apr 04, 2021.

[12] 2006. CVE-2006-4790. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4790. Accessed: Apr 04, 2021.

[13] 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: Apr 04, 2021.

[14] 2021. AFL++ – The AFL++ fuzzing framework. https://aflplus.plus. Accessed: Apr 04, 2021.

[15] 2021. ClusterFuzz – Scalable Fuzzing Infrastructure. https://github.com/google/clusterfuzz. Accessed: Apr 04, 2021.

[16] 2021. FuzzBench – Fuzzer benchmarking as a service. https://github.com/google/fuzzbench. Accessed: Apr 04, 2021.

[17] 2021. Grammar Mutator – AFL++. https://github.com/AFLplusplus/Grammar-Mutator. Accessed: Apr 04, 2021.

[18] 2021. Honggfuzz – Security oriented software fuzzer. https://github.com/google/honggfuzz. Accessed: Apr 04, 2021.

[19] 2021. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: Apr 04, 2021.

[20] 2021. Morpheus – A PKCS1 signature verification non-compliance checker. https://github.com/Morpheus-Repo/Morpheus.git. Accessed: May 04, 2021.

[21] 2021. OSS-Fuzz – Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: Apr 04, 2021.

[22] 2021. Peach – Peach Fuzzer. http://www.peachfuzzer.com/. Accessed: Apr 04, 2021.

[23] 2021. SPIKE – Fuzzer Automation with SPIKE. https://resources.infosecinstitute.com/topic/fuzzer-automation-with-spike/. Accessed: Apr 04, 2021.

[24] Alberto Bartoli, Eric Medvet, and Filippo Onesti. 2018. Evil twins and WPA2 Enterprise: A coming security disaster? *Computers & Security* 74 (2018), 1–11.

[25] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy*.

[26] Daniel Bleichenbacher. 1998. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, Hugo Krawczyk (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

[27] Hanno Böck, Juraj Somorovsky, and Craig Young. 2018. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 817–849. https://www.usenix.org/conference/usenixsecurity18/presentation/bock

[28] George E. P Box. 2005. *Statistics for experimenters : design, innovation and discovery* (2nd ed. ed.). Wiley-Interscience, Hoboken, N.J.

[29] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 114–129.

[30] Bugzilla. 2014. RSA PKCS#1 signature verification forgery is possible due to too-permissive SignatureAlgorithm parameter parsing. https://bugzilla.mozilla.org/show_bug.cgi?id=1064636. Accessed: Apr 04, 2021.

[31] Bugzilla. 2014 (accessed Feb 08, 2021). *RSA PKCS#1 signature verification forgery is possible due to too-permissive SignatureAlgorithm parameter parsing.* https://bugzilla.mozilla.org/show_bug.cgi?id=1064636.

[32] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. 503–520. https://doi.org/10.1109/SP.2017.40

[33] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS# 1 v1. 5 Signature Verification. In *NDSS*.

[34] The Coq Development Team. 2012. *The Coq Reference Manual, version 8.12.* Available electronically at https://coq.inria.fr/distrib/current/refman/.

[35] Siddhartha R. Dalal and Colin L. Mallows. 1998. Factor-Covering Designs for Testing Software. *Technometrics* 40, 3 (Aug. 1998), 234âĂŞ243. https://doi.org/10.2307/1271179

[36] Antoine Delignat-Lavaud, Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, and Yinglian Xie. 2014. Web PKI: Closing the Gap between Guidelines and Practices.. In *NDSS*. Citeseer.

[37] D. Eastlake. 2001. RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS). RFC 3110. https://www.rfc-editor.org/rfc/rfc3110.txt

[38] H. Finney. 2006. Bleichenbacher's RSA signature forgery based on implementation error. https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/. Accessed: Apr 04, 2021.

[39] Si Gao, Hua Chen, and Limin Fan. 2013. Padding Oracle Attack on PKCS#1 v1.5: Can Non-standard Implementation Act as a Shelter?. In *Cryptology and Network Security*. Springer International Publishing, Cham, 39–56.

[40] Tibor Jager, Saqib A. Kakvi, and Alexander May. 2018. On the Security of the PKCS#1 v1.5 Signature Scheme *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1195âĂŞ1208. https://doi.org/10.1145/3243734.3243798

[41] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. 2012. Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption. In *Computer Security – ESORICS 2012*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 752–769.

[42] Simon Josefsson. 2006. [gnutls-dev] Original analysis of signature forgery problem. https://lists.gnupg.org/pipermail/gnutls-dev/2006-September/001240.html. Accessed: Apr 04, 2021.

[43] Simon Josefsson. 2006 (accessed Feb 08, 2021). *[gnutls-dev] Original analysis of signature forgery problem.* https://lists.gnupg.org/pipermail/gnutls-dev/2006-September/001240.html.

[44] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. [n.d.]. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. ([n. d.]).

[45] Burt Kaliski. 1998. PKCS #1: RSA Encryption Version 1.5. RFC 2313. https://doi.org/10.17487/RFC2313

[46] Burt Kaliski and Jessica Staddon. 1998. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437. https://doi.org/10.17487/RFC2437

[47] Burt Kaliski and Jessica Staddon. 2003. RFC3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447. https://doi.org/10.17487/RFC3447

[48] Cameron F. Kerry and Charles Romine. 2013. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS).

[49] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[50] Vlastimil Klíma, Ondrej Pokorný, and Tomáš Rosa. 2003. Attacking RSA-Based Sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, Colin D. Walter, Çetin K. Koç, and Christof Paar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 426–440.

[51] D. Richard Kuhn and Raghu N. Kacker. 2011. Combinatorial Testing. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=910001. Accessed: Apr 04, 2021.

[52] Ulrich Kühn, Andrei Pyshkin, Erik Tews, and Ralf-Philipp Weinmann. 2008. Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures. In *Sicherheit 2008: Sicherheit, Schutz und Zuverlässigkeit. Konferenzband der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 2.-4. April 2008 im Saarbrücker Schloss.*

[53] Ulrich Kühn, Andrei Pyshkin, Erik Tews, and Ralf-Philipp Weinmann. 2008. Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures. In *SICHERHEIT 2008 âĂŞ Sicherheit, Schutz und Zuverlässigkeit. Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)*, Ammar Alkassar and Jörg Siekmann (Eds.). Gesellschaft für Informatik e. V., Bonn, 97–109.

[54] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21). USENIX Association.*

[55] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. 2014. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security*

Symposium (USENIX Security 14). USENIX Association, San Diego, CA, 733–748. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer

[56] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. 2016. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017. https://doi.org/10.17487/RFC8017

[57] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632. https://doi.org/10.1109/SP.2017.27

[58] Tahina Ramananandro, Antoine Delignat-Lavaud, CÃĺdric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *USENIX Security*. USENIX. https://www.microsoft.com/en-us/research/publication/everparse/

[59] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. 2019. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *2019 IEEE Symposium on Security and Privacy (SP)*. 435–452. https://doi.org/10.1109/SP.2019.00062

[60] Joseph A. Salowey, Sean Turner, and Christopher A. Wood. [n.d.]. TLS 1.3: - One Year Later. https://www.ietf.org/blog/tls13-adoption/. Accessed: Jan 11, 2020.

[61] George B. Sherwood. 2015. Embedded functions in combinatorial test designs. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICSTW.2015.7107432

[62] N. J. A. Sloane. 1993. Covering arrays and intersecting codes. *Journal of Combinatorial Designs* 1, 1 (1993), 51–63. https://doi.org/10.1002/jcd.3180010106 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcd.3180010106

[63] Nigel P. Smart. 2015. *Cryptography Made Simple* (1st ed.). Springer Publishing Company, Incorporated.

[64] Serge Vaudenay. 2010. *A Classical Introduction to Cryptography: Applications for Communications Security* (1st ed.). Springer Publishing Company, Incorporated.

A EVOLUTION OF PKCS#1-V1.5 STANDARD

RFC8017 [56] describes the latest version of RSA Cryptography Specifications in which it provides recommendations for the implementation of public-key cryptography based on the RSA algorithm. There are two types of schemes described in the PKCS#1 specifications: *encryption* and *signature with appendix*. These schemes are to combine cryptographic primitives and other techniques in order to achieve a particular security goal. The signature type consists of two schemes, called `RSASSA-PKCS1-v1_5` and `RSASSA-PSS` , where the former has been maintained from the earliest version (v1.5) and the latter is introduced in the interest of increased robustness. According to RFC8017, `RSASSA-PKCS1-v1_5` (dubbed as PKCS#1-v1.5) is still appropriate for new applications because no attack has been found to the specifications. In fact, the PKCS#1-v1.5 signature scheme is the most widely used digital signature in practice [40]. The popularity is because that it appears to be simple, and thus (seemingly) easy to implement, while featuring significantly faster operations than its peers, such as DSA or ECDSA [40]. Another reason that boosts its popularity is its widely adoption in some protocols which are still ubiquitous on the Internet [60].

PKCS#1-v1.5 signature scheme was born in RFC2313 [45] where it specifies the primitives for RSA signature generation and signature signature verification. The encoded message format was introduced in this RFC, while for the ASN.1 data portion, it was suggested to use BER (Basic Encoding Rules [10]). The hash functions recommended were `MD2`, `MD4`, and `MD5` and for verification operation, it was suggested to use decode approach to parse BER in order to extract hash function ID and hash value.

In the subsequent RFC2437 [46] (which obsoletes RFC2313), a more organized and detailed specifications of the signature generation and verification primitives are given for PKCS#1-v1.5 signature scheme where it updates the ASN.1 data portion of the encoded message to follow DER. This RFC also adds `SHA-1` to the recommended list of hash functions and removes `MD4` from it (flagged as being vulnerable). As specified in this RFC, the parameter field for associated to the hash function in an `AlgorithmIdentifier` shall have type `NULL`.

The RFC3447 [47] (which obsoletes RFC2437) introduces a new signature scheme, called `RSASSA-PSS`, although no attack has found in PKCS#1-v1.5, and emphasizes it is still appropriate for the new applications to use PKCS#1-v1.5 signature scheme. This RFC also suggests a cautious implementor to support BER encoding as long as the overall structure is correct. It then describes the encoding-based and decoding-based approaches are two alternatives for the verification, while not promoting one over the other. This RFC also adds three more hash functions, `SHA-256`, `SHA-384`, and `SHA-512` to the previous recommended list and suggest the parameter field associated to `SHA` hash function family "should be omitted, but if present, shall have a value of type NULL".

Finally, in RFC8017 [56] (the latest version to the date), although keeping the most of specifications from its predecessor, it adds support for three more hash functions: `SHA-224`, `SHA-512/224`, and `SHA-512/256` and recommends `SHA-2XX` family onwards. RFC8017 also emphasizes that "for the SHA algorithms, implementations MUST accept `AlgorithmIdentifier` values both without parameters and with `NULL` parameters". However, it mentions in formatting the `DigestInfo` to generate the encoded message (during signature generation operation), the parameters field associated with `SHA` hash function family shall have a value of type NULL to maintain compatibility with existing implementations.

B PARAMETER SETTINGS IN MORPHEUS

Concrete values used in the evaluation. Following from the reference notation introduced in Table 1 and the input parameters discussed in Section 5.4, we have used the following concrete values in our evaluation of Morpheus.

```
min, max  = 1, 2
C's labels  = ['leading_byte', 'block_type', 'padding_bytes', 'pa
     dding_end', 'asb@type', 'asb@length', 'hash_algo@type', 'has
     h_algo@length', 'hash_id@type', 'hash_id@length', 'hash_id@v
     alue', 'param@type', 'param@length', 'param@value','hash_val
     ue@type', 'hash_value@length', 'hash_value@value']
S's labels = C's labels
n  = 0xE932AC92252F585B3A80A4DD76A897C8B7652952FE788F6EC8DD640587
     A1EE5647670A8AD4C2BE0F9FA6E49C605ADF77B5174230AF7BD50E5D6D6D
     6D28CCF0A886A514CC72E51D209CC772A52EF419F6A953F3135929588EBE
     9B351FCA61CED78F346FE00DBB6306E5C2A4C6DFC3779AF85AB417371CF3
     4D8387B9B30AE46D7A5FF5A655B8D8455F1B94AE736989D60A6F2FD5CADB
     FFBD504C5A756A2E6BB5CECC13BCA7503F6DF8B52ACE5C410997E98809DB
     4DC30D943DE4E812A47553DCE54844A78E36401D13F77DC650619FED88D8
     B3926E3D8E319C80C744779AC5D6ABE252896950917476ECE5E8FC27D5F0
     53D6018D91B502C4787558A002B9283DA7
|n|= 256 bytes
d  = 0x009b771db6c374e59227006de8f9c5ba85cf98c63754505f9f30939803
     afc1498eda44b1b1e32c7eb51519edbd9591ea4fce0f8175ca528e09939e
     48f37088a07059c36332f74368c06884f718c9f8114f1b8d4cb790c63b09
     d46778bfdc41348fb4cd9feab3d24204992c6dd9ea824fbca591cd64cf68
     a233ad0526775c9848fafa31528177e1f8df9181a8b945081106fd58bd3d
     73799b229575c4f3b29101a03ee1f05472b3615784d9244ce0ed639c77e8
     e212ab52abddf4a928224b6b6f74b7114786dd6071bd9113d7870c6b52c0
     bc8b9c102cfe321dac357e030ed6c580040ca41c13d6b4967811807ef2a2
     25983ea9f88d67faa42620f42a4f5bdbe03b
e  = 3
H  = SHA-256 (OID = 0x608648016503040201)
m  = "hello world!"
```

C COMPARING MORPHEUS WITH PREVIOUS WORK [33]

The results of comparing Morpheus with respect to the prior by Chau *et al.* [33] can be found in Table C1 and Table C2.

Table C1: Comparing Morpheus and previous work [33] on previously tested subjects

Implementation name	Source code language	Bug[1]	Chau *et al.* [33]	Morpheus
axTLS v2.1.3	C	SF#1: Accepting trailing bytes	✓	✓
		SF#2: Ignoring prefix bytes	✓	✓
		SF#3: Ignoring ASN.1 metadata	✓	✓
		BO#1: Trusting declared lengths	✓	✓
		ML#1: Accepting definite long form encoding of length for both digestAlgorithm and digest	X	✓
libtomcrypt v1.16	C	SF#1: Accepting trailing bytes	✓	✓
		SF#2: Accepting less than 8 bytes of padding	✓	✓
		ML#1: Improper checking of digestAlgorithm length	✓	✓
		ML#2: Ignoring form bit of type octet for digest and digestAlgorithm's identifier	X	✓
		IN	X	✓
GnuTLS v1.4.2[2]	C	SF#1: Ignoring digestAlgorithm's identifier value octets	✓	✓
		SF#2: Ignoring digestAlgorithm's parameter value octets	✓	✓
		SF#3: Ignoring the padding string	✓	✓
		ML#1: Ignoring digestAlgorithm's parameter type octet	X	✓
		ML#2: Accepting definite long form encoding of length for DigestInfo	X	✓
MatrixSSL v3.9.1 (Certificate)	C	ML: Lax ASN.1 length checks	✓	✓
		IN	X	✓
MatrixSSL v3.9.1 (CRL)	C	ML#1: Lax ASN.1 length checks	✓	✓
		ML#2: Mishandling Algorithm OID	✓	✓
		IN	X	✓
mbedTLS v2.4.2	C	ML: Lax algorithm parameter length check	✓	✓
		IN	X	✓
openssl v0.9.7h[2]	C	SF: Accepting trailing bytes	✓	✓
		ML: Accepting some absurd length values	✓	✓
Openswan v2.6.50	C	SF: Ignoring padding bytes	✓	✓
		IN	X	✓
strongSwan v5.6.3	C	SF#1: Not checking algorithm parameter	✓	✓
		SF#2 Accepting trailing bytes after OID	✓	✓
		SF#3: Accepting less than 8 bytes of padding	✓	✓
		ML: Lax ASN.1 length checks	✓	✓

[1] SF: Signature Forgery; ML: Minor Leniency; BO: Buffer Overflow; IN: INcompatibility issue
[2] Libraries used for feasibility study in [33]

Table C2: Statistics of applying the KLEE toolchain from previous work [33] on new test subjects

Implementation (version)	Test Harness	Lines Changed	Execution Time	Total Path (Accepting)
Apache milagro v2.0.1	TH1	0	<9 mins	60582 (1)
	TH2		<1 min	42 (1)
	TH3		<1 min	6 (1)
RELIC git commit 32eb4c25	TH1	8	<26 mins	20595 (15)
	TH2		<3 mins	36 (3)
	TH3		<1 min	2 (1)
wpa_supplicant v2.9	TH1	11	<2 mins	2685 (3)
	TH2		<1 min	1224 (21)
	TH3		<1 min	161 (2)

D DETAILED FINDINGS

Here we present detailed descriptions and root cause analysis for some of our findings, and provide sample encoded messages `EM` and signatures `S` demonstrating the unwarranted leniency. Each signature `S` given here was generated and can be verified using the modulus and exponents given in Appendix B. More details can be found in [20].

D.1 node-forge (v0.10.0)

D.1.1 Accepting less than 8 bytes of padding [ML#1]. In the line 1575 in `_decodePkcs1_v1_5()` function from `node_forge/lib/rsa.js`, the implementation does not check the padding bytes minimum required length of 8 or more, instead, it looks for the first byte that is not `0xFF` and then counts the number of padding bytes (*i.e.*, using `padNum` variable). It then checks that the end of padding, `zero` variable, is actually `0x00` and the number of padding bytes is (k − 3−`T`'s length), where `k` is the length of public modulus `n` (*i.e.*, `|n|`). However, the padding bytes length check performed in code (line 15) can be bypassed because it does not validate whether the top ASN.1 encoded structure (*i.e.*, `T`) is malformed or not, and mistakenly trusts whatever it contains. As we show in other vulnerabilities, this bug enables attacker to steal all bytes from padding bytes and use them to expand an unchecked portion size of the encoded message structure (because of some existing leniencies) to launch Bleichenbacher-style low public exponent RSA signature forgery.

```
1  } else if(bt === 0x01) {
2    // find the first byte that isn't 0xFF, should be
         after all padding
3    padNum = 0;
4    while(eb.length() > 1) {
5      if(eb.getByte() !== 0xFF) {
6        --eb.read;
7        break;
8      }
9      ++padNum;
10   }
11 }
12 ...
13 // zero must be 0x00 and padNum must be (k - 3 -
        message length)
14 var zero = eb.getByte();
15 if(zero !== 0x00 || padNum !== (k - 3 - eb.length())
       ) {
16   throw new Error('Encryption block is invalid.');
17 }
18 return eb.getBytes();
```

D.1.2 Ignoring digestAlgorithm structure (CVE-2021-30247) [SF#1]. The `DigestInfo` is the top ASN.1 encoded sequence structure that contains `digestalgorithm` and `digest` octet strings, where the former encodes the hash ID information, being used in the signing process as well as the optional parameter field associated to that hash algorithm (represented by NULL parameter), and latter encodes octet string TLV containing the actual hash value of the message being signed. The implementation in node-forge, however, is lenient in checking the `digestAlgorithm` structure. Once `DigestInfo` is being decoded, the implementation, does not check the necessary elements to be present in its first child (*i.e.*, `digestalgorithm`) and just retrieves the hash value from its second child (*i.e.*, `digest`) to compare it against the computed hash value. Hence, it is possible to build arbitrary TLV and use that instead of the `digestAlgorithm` as long as the TLV is well-formed.

Now by taking a closer look at line 11 in `verify()` function snippet below taken from `node_forge/lib/rsa.js`, we can spot that the implementation only checks that the hash value from the decoded top ASN.1 structure is equal to the given digest. However, it mistakenly ignores checking the other elements in the `digestAlgorithm` structure, decoded into `obj.value[0]`. These missing checks are to make sure the hash function used to compute `digest` argument is the same as the hash ID encoded in `digestAlgorithm`. Also, the parameter field associated to the hash function encodes `NULL` TLV. Such checks are necessary to avoid attacker from using these areas to launch Bleichenbacher-style low public exponent RSA signature forgery.

```
1  key.verify = function(digest, signature, scheme) {
2    ...
3    if(scheme === 'RSASSA-PKCS1-V1_5') {
4      scheme = {
5        verify: function(digest, d) {
6          // remove padding
7          d = _decodePkcs1_v1_5(d, key, true);
8          // d is ASN.1 BER-encoded DigestInfo
9          var obj = asn1.fromDer(d);
10         // compare the given digest to the decrypted
               one
11         return digest === obj.value[1].value;
12       }
13     };
14   ...
15   // do rsa decryption w/o any decoding, then verify
       -- which does decoding
16   var d = pki.rsa.decrypt(signature, key, true,
         false);
17   return scheme.verify(digest, d, key.n.bitLength())
         ;
18 };
```

In order to reproduce triggering this bug, the below concrete values, found by Morpheus, can be used given the parameter settings provided in section B.

Example: `91` garbage bytes injected as the value of a TLV replaced `digestAlgorithm` structure.

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffff00307f065b888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888804207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0xe7410e05bdc38d1c72fab784be41df3d3de2ae83894d9ec86cb5fe343d
     5dc7d45df2a36fc60363faf32f0d37ab457648af40a48a6c53ae7af0575e
     92cb1ffc236d55e1325af8c71b3ac313f2630fb498b8e1546093aca1ed56
     026a96cb525d991159a2d6ccbfd5ef63ae718f8ace2469e357ccf3f6a048
     bbf9760f5fb36b9dd38fb330eab504f05078b83f5d8bd95dce8fccc6b46b
     abd56f678300f2b39083e53e04e79f503358a6222f8dd66b561fea3a51ec
     f3be16c9e2ea6ba8aaed9fbe6ba510ff752e4529385f759d4d6120b15f65
     534248ed5bbb1307a7d0a9838329697f5fbae91f48e478dcbb77190f0d17
     3b6cb8b1299cf4202570d25d11a7862b47
```

D.1.3 Accepting trailing bytes (CVE-2021-30249) [SF#2]. This leniency stems from the mistake in the implementation where after reading the length of `DigestInfo` and decoding its nested TLV structure with respect to that length, the code does not check that there should not be any trailing bytes left while all content octets of the `DigestInfo` have been decoded. As can be seen in line 1 of `_fromDer()` function snippet taken from `node_forge/lib/asn1.js`, once length octet for the top ASN.1 structure is being decoded, it is being used in a subsequent while loop to decode the nested ASN.1 structure. When the `length` value reaches `0`, the implementation finishes the decoding of ASN.1 structure and returns it to the callee, `fromDer()` function, which in turn gives it to `verify()` function. However, in none of these steps, the implementation does not check that there is no garbage bytes trailing the ASN.1 structure, and thus will be ignored during the process.

```
1  var length = _getValueLength(bytes, remaining);
2  ...
3  var constructed = ((b1 & 0x20) === 0x20);
4  if(constructed) {
5    // parse child asn1 objects from the value
6    value = [];
7    if(length === undefined) {
8      ...
9    } else {
10     // parsing asn1 object of definite length
11     while(length > 0) {
12       start = bytes.length();
13       value.push(_fromDer(bytes, length, depth + 1,
             options));
14       remaining -= start - bytes.length();
15       length -= start - bytes.length();
16     }
17   }
18 }
19 ...
20 // create and return asn1 object
21 return asn1.create(tagClass, type, constructed,
       value, asn1Options);
```

In order to reproduce triggering this bug, the below concrete values, found by Morpheus, can be used given the parameter settings provided in section B.

Example: `215` garbage bytes added as trailing garbage bytes by exploiting another vulnerability which ignores `digestAlgorithm` structure to expand the size of unchecked trailing garbage bytes.

```
EM = 0x0001003024010004207509e5bda0c762d2bac7f90d758b5b2263fa01cc
     bc542ab5e3df163be08e6ca9888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888888888888
S  = 0xa7c5812d7fc0eef766a481aac18c8c48483daf9b5ffb6614bd98ebe4ec
     b746dd493cf5dd2cbe16ecaa0b52109b744930eda49316605fc823fd57a6
     8b5b2c62e8c1b158b26e1547a2e33cdd79427d7c513f07d02261ffe43db1
     97d8cddca2b5b43c1df85aaed6e91aadd44a46bff7f5c70f1acc1a193917
     e3908444632f30e69cfe95d8036d3b6ad318eefd3952804f16613c969e6d
     13604bb4e723dfad24c42c8d9b5b16a9f5a4b40dcf17b167d319017740f9
     cc0836436c14d51c3d8a697f1fa2b65196deb5c21b1559c7dea7f598007f
     a7320909825009f8bf376491c298d8155a382e967042db952e995d14b2f9
     61e1b22f911d1b77895def1c7ef229c87e
```

D.1.4 Leniency in checking type octet [ML#2]. This issue happens for the lack of necessary checks for each type octet in T. Although the implementation decodes the type information,they are not actually being checked to match with the expected types in the TLV structures within T. This minor leniency does not lead to any signature forgery; however, mistakenly accepting an invalid signature can create interoperability issue.

Example: Incorrect value (*i.e.*, `0x0c`) used for `digestAlgorithm`'s type octet instead of the correct value of `0x30`:

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffff0030310c0d
     060960864801650304020105000420750 9e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0xd8298a199e1b6ac18f3c0067a004bd9ff7af87be6ad857d73cc3d24ef0
     6195b82aaddb0194f8e61fc31453b9163062255e8baf9c480200d0991a5f
     764f63d5f6afd283b9cd6afe54f0b7f738707b4eb6b8807539bb627e74db
     87a50413ab18e504e37975aad1edc612bc8ecad53b81ea249deb5a2acc27
     e6419c61ab9acec6608f5ae6a2985ba0b6f42d831bc6cce4b044864154b9
     35cf179967d129e0ad8eda9bfbb638121c3ff13c64d439632e62250d4be9
     28a3deb112ef76a025c5d918051e601878eac0049fc9d82be9ae3475deb7
     ca515c830c20b91b7bedf2184fef66aea0bde62ccd1659afbfd1342322b0
     95309451b1a87e007e640e368fb68a13c9
```

D.2 wpa_supplicant & hostapd (v2.9)

D.2.1 Lax length octet checking for DER [ML#1]. In ITU-T X.690 standard [10], three ways for encoding length are introduced: *definite short form*; *definite long form*; and *indefinite form*. Definite short form uses one octet for the length value in the range [0, 127]. Definite long form uses an initial octet followed by one or more subsequent octets. The initial octet shall be encoded as 8th bit (MSB) is 1 while bits 7 to 1 shall encode the number of subsequent octets in the length octets, as an unsigned binary integer with bit 7 as the most significant bit. Bits 8 to 1 of the first subsequent octet, followed by bits 8 to 1 of the second subsequent octet, followed in turn by bits 8 to 1 of each further octet up to and including the last subsequent octet, shall be the encoding of an unsigned binary integer equal to the number of octets in the contents octets, with bit 8 of the first subsequent octet as the most significant bit. For the indefinite form, the length octets indicate that the contents octets are terminated by end-of-contents octets, and shall consist of a single octet. However, as in section 10.1 of DER encoding rule [10] mandates:

> "the definite form of length encoding shall be used, encoded in the minimum number of octets."

That is, if a length value is less than or equal 127 bytes, it must be of definite short form, while definite long form is only allowed for 128 bytes or larger. That being said, the two implementations mistakenly validate signature values whose ASN.1 structure does not follow DER for encoding the length octets. For example, for a length of 9 bytes, instead of using only definite short form to encode it within one byte as `0x09`, these implementations also allow it to be encoded as `0x83000009`, where `0x83` is the initial byte indicating that the length is encoded in the three subsequent octets and `0x000009` shows those three subsequent octets representing the length value of 9 bytes.

Root causing the issue, takes us to line 1 of below snippet from `asn1_get_next()` function in `asn1.c` file, where it checks whether or not the length octet uses definite long form. If so, it reads the whole length octets in a loop but it does not check the length encoded in definite long form to be 128 bytes or greater.

```
1  if (tmp & 0x80) {
2     if (tmp == 0xff) {
3        wpa_printf(MSG_DEBUG, "ASN.1: Reserved length "
4              "value 0xff used");
5        return -1;
6     }
7     tmp &= 0x7f; /* number of subsequent octets */
8     hdr->length = 0;
9     if (tmp > 4) {
10       wpa_printf(MSG_DEBUG, "ASN.1: Too long length
               field");
11       return -1;
12    }
13    while (tmp--) {
14       if (pos >= end) {
15          wpa_printf(MSG_DEBUG, "ASN.1: Length "
16                "underflow");
17          return -1;
18       }
19       hdr->length = (hdr->length << 8) | *pos++;
20    }
21 } else {
22    /* Short form - length 0..127 in one octet */
23    hdr->length = tmp;
24 }
```

This bug alone can cause an interoperability issue where an invalid signature value can be mistakenly accepted as valid. However, allowing definite long form, even for its correct length values, will enable attacker to expand the unchecked area, if any, especially when in none of the supported hash functions we do not have any content octets in `T` whose length is greater than 127 bytes.

In order to reproduce triggering this bug, the below concrete values, found by Morpheus, can be used given the parameter settings provided in section B.

Example: Incorrect encoding of the length octet of OID in `AlgorithmIdentifier` as definite long form with 4 bytes (`0x83000009`):

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffff0030343010068300
     0009608648016503040201050004207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0x80e586e59f6294c9b177a0cf256abf7b91916baa08b3d833b80b60bf0c
     85fb3c7d8bcc98eeaf94277e1cd8a1802c706554115b5342f7d96f6cc753
     69616a378b78708e40b1e9582213ee10cf2893e197f89256e88dc00bf6cf
     17513cdd296b41b5aeb60e5fc609614cee18bd751c6e22d668962395e112
     d1938501d62dd5e78ddad458d55ddcf7a9cbd8dd8792eedaed27172bcb95
     b817f360103b879f01be63572a9b03b9912759b1e503e4ed96e285b13b5e
     8aa00898c1ee7c58915673ea7dc566243b376c14711773f8bd955e5300df
     e8528d2a8d71063579a5813491c9be3058a317b496364021169df10a1486
     16862f58ff10401efa1300dbbc114704f7
```

D.2.2 Leniency in checking `AlgorithmIdentifier` structure (CVE-2021-30004) [SF#1]. The logics in these implementations follow these steps. After modular exponentiation to the power `e`, and having EM_v at hand, it starts reading the TLVs in `T`. While reading TLVs from top to bottom, once it reaches to `AlgorithmIdentifier` and reading off hash function's `OID`, it skips what follows (see line 11 from the below snippet of `x509_check_signature()` function in `x509v3.c` file) — the explicit `NULL` hash algorithm parameter TLV — and then starts checking the necessary steps before reading/matching the hash value portion. So, the root cause is that not sufficient check is taken and the explicit `NULL` parameter section of EM_v is being skipped mistakenly, assuming that it is not important to check. This leniency in checking the explicit `NULL` parameter TLV enables an attacker to exploit this unchecked area to launch Bleichenbacher-style low public exponent RSA signature forgery.

```c
if (x509_sha256_oid(&oid)) {
   if (signature->oid.oid[6] !=
       11 /* sha2561WithRSAEncryption */) {
      wpa_printf(MSG_DEBUG, "X509: digestAlgorithm
           SHA256 "
            "does not match with certificate "
            "signatureAlgorithm (%lu)",
            signature->oid.oid[6]);
      os_free(data);
      return -1;
   }
   goto skip_digest_oid;
}
```

Reproducing this bug can be done using the below concrete values, found by Morpheus, given the parameter settings provided in section B.

Example: 194 garbage bytes (`0x88`s) are injected in the explicit `NULL` parameter TLV of `AlgorithmIdentifier` structure:

```
EM = 0x0001ffffffffffffffff003081f13081cd060960864801650304020188
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888804207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0x565e9302375fa303761eaf92359597b06f8781e31e1fc2b86ec273e211
     dab37c3afe2fd7c9d8543d2b45e8a98a38f3bc9536165d94a133b3afba58
     c934e85afb99e46d0216a61c72f1323d5cbe833be8b750e2b879d0217c4d
     b269771fdf9bf865907f3101cd5aa227fd824e35ecc2e3ec90e64633abc2
     cf039d7e13f87cd1f2e06ffb78e28c25482f23983a30d2ae02513fc0c8bb
     1905f09bb593df412b810a0f807cd7d9fa25b8323fe3537c39d21dc76a3c
     cf22428243d3c44f0cef005fe6e26f01f3a2f546c11c0deb48f8e6a6954a
     ea8d68336fa8378458f2e4c84217a6419be0f7423edfcf9ac2bb3d6e6854
     872e01057faed6a89932fc6c0a76f07c31
```

D.2.3 *Leniency in checking form bit of an identifier octet [ML#2].*

An identifier octet in DER encoding [10] of a data value consists of three components: *Class* (two most significant bits); *Form* (6th bit, indexing from 1); and *Tag* (five least significant bits). The Class component is there to indicate the class of a Tag, whether it is a standard type or application specific type. For instance, `00` for Class bits shows the tag is universal type. The Form component indicates if the tag does include sub-type, and thus represents a constructed type (if set), or it is just a primitive type without any sub-type (if not set). Finally, Tag represents the actual type of the data value in DER encoding. For example, the tag for a `sequence` type is `b'10000'`; including Form bit to that makes it `b'110000'` because `sequence` tag is constructed not primitive; and given that the `sequence` type has universal tag class we end up having `b'00110000'` (`0x30` in hex) as the final encoded value for this identifier octet. In both implementations, the Form bit, however, is not being checked and thus it accepts two versions for an identifier octet (*i.e.*, with and without Form bit being set). That being said, an identifier octet like `sequence` can be represented as both `b'00110000'` (*i.e.*, `0x30`) and `b'00010000'` (*i.e.*, `0x10`). This issue affects `sequence` type used in the top DER encoded ASN.1 structure (*i.e.*, `DigestInfo`); `sequence` type in `AlgorithmIdentifier` structure; `object identifier` type of hash ID encoding structure; and `octet string` type of hash value TLV structure.

Example: Incorrect encoding of `sequence` type (`0x10` instead of the correct encoding `0x30`) used in the `DigestInfo` structure is accepted as a valid signature by these implementations.

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffff001031300d
     0609608648016503040201050004207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0xbd1f63e0846661cbc98dd389ea2d7fedc3d74f133a7741f2cde00d71e3
     75e89155522f7c3ef1035464e0082ddd5df3244ce0b59f248624b942bb5a
     4032262ca9ef96e6c2e81cca72716640ae692f71db26a7736ffa0986217e
     688a42580d85aa90a3eb318bf53c7ade07379662414eccb9cbd985779cc0
     da6e1f495bb5e899fa2276974b5e5c8fee3615f1cefeb2911e3a6876849e
     1b224cec26d09112f9c54f0c6daf9f8b070acd53f128d0fbd79a7a81d19a
     3d6e09464e1b3da2aaec410f8e883b45cd003c74515144f9b8dffd0ecfa2
     146342578f1e1d5bba9f47ce8f3e57372420a1f1bb458246a4cbd756f8bd
     d5d86f7294031f178d3844e7f997b40993
```

D.2.4 *Incorrect extended tag decoding of identifier octets [ML#3].*

According to the section 8.1.2.4 of ITU-T X.690 [10] standard specifying ASN.1 encoding format, for

> "tags with a number greater than or equal to `31`, the identifier shall comprise a leading octet followed by one or more subsequent octets."

This allows tag to be extended to support various types with higher tag numbers. For that, we have a sequence of bytes starting from leading byte, continuing by subsequent octets, as needed, followed by the last octet. These bytes need to conform the encoding rules prescribed by ITU-T X.690 standard but some important ones are as follows. The first 5 least significant bits (tag component) of the leading byte must be all 1s (`0x1f`). Then each subsequent octets (except the last octet) must have the most significant bit set to 1 while the last octet's MSB must be `0`. Now once these bytes are read, the extended tag is computed by concatenating all subsequent octets (including the last octet) such that all MSBs (for each byte) are eliminated. The issue with these implementations is that even though it deals with a primitive type, like `object identifier` type (`0x06`), it allows one to have an extended tag version of that as well (*e.g.*, `0x1f06`) which is against the standard that prescribes the extended tags which emphasizes "for tags with a number greater than or equal to 31". Besides, even though the standard, in the section 8.1.2.4.2.c, prohibits that "bits 7 to 1 of the first subsequent octet shall not all be zero", these implementations are non-compliant. Therefore, one can create arbitrary large extended tag format. Going back to the `object identifier` type example, we can have `0x1f808080...8006` and still be recognized as a valid identifier octet for the object identifier type.

Although this bug does not lead to any signature forgery, it can cause interoperability issue by accepting some invalid signatures.

Example: Incorrect encoding of `object identifier` type's identifier octet as the extended tag with two bytes (`0x1f06`):

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffff003031300d
     1f06096086480165030402010504207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
```

```
S  = 0xdf0ddee8731d914b99320cda04c2c84c637a3fb8c5b4155af69ba58b81
       c352c4ae4e88093265b8f0d83f4597e5fa0e07ec2637a276101f9f40275d
       70efb792132dd66dca80a2667ed9e9b3f1781236fe1022d19c81c3a41e0c
       0e3b222a4c429bbf35b60a68b9d7fc121451e657041bf8a056e7ef9c281f
       2e73636f69a457dca3bdc3d142794220bbd4ac257969d727bd90673a03d2
       5f887361f782f610ce4d1ca4ffe2b6023a183a68026b8e9bc758a9d7076e
       b51840a9006b86477bc69273e1cfb119e0670b10b8d7144262695d250e81
       5c64961b155d0c51d562f8992b1ac47cad9c1c395692736a7f8653bbf43d
       3e04e9f6a1ac11b1d8f0d9357349776a82
```

D.2.5 Leniency in checking the length octet of `DigestInfo` *[ML#4].*

The way an ASN.1 encoding structure (identifier octets (Type); length octets (Length); contents octets (Value), TLV for short) is being parsed in these implementations is that, simply put, it takes a chunk of data, with its length and then parses it to return a pointer to its header (type section of TLV). To do so, it considers the first byte as header, reads the second byte as length section, and lastly checks if the current position (which now refers to the first byte of value section) has not passed the end position (previously computed from the data length input). It also checks whether the length field of a TLV is not greater than what has remained to read (i.e., `end - pos`). So, as long as the value for the length section of a TLV is not greater than what has left to read, everything checks out; even though the value is less than the correct value. The length field for `DigestInfo` TLV suffers from this leniency; so, lower value than the correct length will also be accepted.

Example: Incorrect value for `DigestInfo`'s length octet as `0x0F` is accepted by the verifier instead of the correct value `0x31`:

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffff
       ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
       ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
       ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
       ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
       ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
       ffffffffffffffffffffffffffffffffffffffffffffffffff00300f300d
       0609608648016503040201050004207509e5bda0c762d2bac7f90d758b5b
       2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0x023d1761649fdc744bbcf7464ae4de054a2a283a0082ed7153325daf81
       13410d112259faa47c5f7c215c369d989cc95e3d6482a4d8e5fe039c6282
       2d5c01c51b51babc97585034360d9397fc375fe185d007d08af2fc80b1d6
       948ebc406cd7a615659b08927a25b80137202dff36034d57e9b9c181306b
       858010e6df04bfc9ddc960cba2b7479279e92b936a3a152ec46320d01380
       70d44a7d98bec5cc3c1dd0de33f589af3480d8894ab61f0d03db4fae97bf
       83defa5c7db2dd11823156b0a0976297efa46e3f86f363af57c4bbb17736
       d5c55d0d5ab92ea062a0dc51294e0071ec26e051ecd1cff4aec04fdcda4a
       6eb1ca1331d7838c7df3ffd5a9039be0a0
```

D.3 RELIC (git commit 32eb4c25)

D.3.1 Leniency in checking the prefix bytes (CVE-2020-36315) [SF#1].

The prefix bytes in the encoded message EM_v consists of the leading byte (with value of `0x00`), block type byte (`0x01` for signature scheme), and padding bytes with an appropriate length to make the whole encoded message length equal to the public modulus length, where every byte of padding bytes has a value of `0xFF` while `0x00` byte is used to indicate the end of padding. The implementation is, however, lenient in parsing these prefix bytes, and thus open the door for a signature forgery attack. `pad_pkcs1()` function in the implementation is responsible to strip off such prefix bytes. The first problem happens because of the way the leading byte and the bock type checks are being handled. As can be seen in the below `pad_pkcs1()` function snippet taken from `src/cp/relic_cp_rsa.c` file, the leading byte and block type are being read and if they are not `0x00` and `0x01`, respectively, the result becomes an error (lines 9 and 15). However, instead of returning right away from the function once the error has been observed, it continues its operation and later on the `result` variable is overwritten in line 38. The second issue which worsen the situation is the next do-while operation on the subsequent bytes for parsing the padding bytes (line 17 to 21). This loop keeps track of the number of bytes left to be read and it reads the encoded message buffer byte by byte. However, according to the loop exit condition, it leaves all padding bytes unchecked and skips them until it sees the end of padding (`0x00`). So, any value, except `0x00`, will be accepted as a padding byte.

```
1   static int pad_pkcs1(bn_t m, int *p_len, int m_len,
        int k_len, int operation) {
2     ...
3     switch (operation) {
4       ...
5       case RSA_VER:
6           m_len = k_len - 1;
7           bn_rsh(t, m, 8 * m_len);
8           if (!bn_is_zero(t)) {
9               result = RLC_ERR;
10          }
11          m_len--;
12          bn_rsh(t, m, 8 * m_len);
13          pad = (uint8_t)t->dp[0];
14          if (pad != RSA_PRV) {
15              result = RLC_ERR;
16          }
17          do {
18              m_len--;
19              bn_rsh(t, m, 8 * m_len);
20              pad = (uint8_t)t->dp[0];
21          } while (pad != 0 && m_len > 0);
22          if (m_len == 0) {
23              result = RLC_ERR;
24          }
25          /* Remove padding and trailing zero. */
26          id = hash_id(MD_MAP, &len);
27          m_len -= len;
28
29          bn_rsh(t, m, m_len * 8);
30          int r = 0;
31          for (int i = 0; i < len; i++) {
32              pad = (uint8_t)t->dp[0];
33              r |= pad - id[len - i - 1];
34              bn_rsh(t, t, 8);
35          }
36          *p_len = k_len - m_len;
37          bn_mod_2b(m, m, m_len * 8);
38          result = (r == 0 ? RLC_OK : RLC_ERR);
39          break;
40        ...
41  return result;
42  }
```

In order to reproduce triggering this bug, the below concrete values, found by Morpheus, can be used given the parameter settings provided in section B.

Example: `202` garbage bytes (`0x88`s) are added before the block type byte (`0x01`) without including padding bytes (`0xFF` bytes).

```
EM = 0x0088888888888888888888888888888888888888888888888888888888
       888888888888888888888888888888888888888888888888888888888888
       888888888888888888888888888888888888888888888888888888888888
       888888888888888888888888888888888888888888888888888888888888
       888888888888888888888888888888888888888888888888888888888888
       888888888888888888888888888888888888888888888888888888888888
       88888888888888888888888888888888888888888888888801003031300d
       0609608648016503040201050004207509e5bda0c762d2bac7f90d758b5b
       2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0xbce78df677f4fa3ab89a4efd02872b3b4f41c75020f08c3e33f2096b56
       ece1bc789b84372a8dfbf450529259a3279f67f2e8020fbdb30055dba8e8
       7f446236c5fb8677c28f47677225a4ae5a4e4f9c8b8b7a6384fdda87858d
```

```
        7eae0a8710a55d57d55c886c4f651abb9fc80be21da9192c6595c54c2900
        6267bf95b67dd57d486af8c83a917dbaf20120d57e6da00a88d097ebbfbf
        60145cad918451be667e5ddc7c1884df9020f291bc1768ba46d31a0b0a47
        a69af912727ca2a5ee04c6b5755efe11a62ec4971e5612cf7dbcd7cada82
        a38d7ddab7584e6a38fc514aedb345168bc8488fddc4c53ab93eab91b457
        f62326fcc0f33a643362aee18636ba3ebd
```

D.3.2 Buffer overflow caused by trailing garbage bytes (CVE-2020-36315) [BO#1]. The implementation does not properly check that after the hash value bytes in T, there are no trailing garbage bytes. According to the implementation of RSA PKCS#1-v1.5 signature verification, `cp_rsa_ver()` function in snippet below, once `pad_pkcs1()` function is called to strip off EM_v prefix, the counter referring to the end of padding is updated such that to indicate the beginning of the hash value portion. Then, all checks are done to make sure the encoded message's ASN.1 related bytes, right before hash value, matches up with the expected bytes (calculated by `hash_id()`). However, neither `pad_pkcs1()` function nor the callee checks that the hash value bytes has no trailing garbage bytes (*i.e.*, the length of hash value bytes are equal to the expected hash length). This trailing garbage bytes can be added by borrowing bytes from the padding bytes to make sure the length of malformed encoded message is not changed. Now after unpadding and knowing the position of hash value bytes, the implementation copies everything from hash value to the end of encoded message into another memory space (pointed by `h1` in the code) to be compared with the computed hash value (pointed by `h2`). Then, the comparison takes place to compare them with respect to the expected hash length, and thus the trailing garbage bytes are ignored, if there is any.

Example: `202` tailing garbage bytes (`0x88`s) are added after the hash value bytes, causing segmentation fault.

```
EM = 0x0001003031300d060960864801650304020105000420750 9e5bda0c762
     d2bac7f90d758b5b2263fa01ccbc542ab5e3df163be08e6ca98888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888888888888
S  = 0x36262822b5089bef18475b37fe61295421c76465cef497440686dc624a
     3f1ed51e82d609211d3754efc785c1b1603259c1f346f54ecea07393088f
     b2c2907c7f3018df2d0610f8492b819fa53b56a503cab1ff730aa2181e01
     d8e4021fef404bc891e13ceedae4d5ddc1f3a5daf6716e6bb527c8bed032
     266f3b3fddee11ae0d805afb768715e5ba78d37e8acef313919f38665e15
     914efb4d8e0031a6c0f59f7d94f11e450edfed50095a3688d1ac28845894
     c642e303a170881b83d2ad0a443bcf2beeabf2d7e2dd8f9761b26e860a5c
     cbb52c05d9558ec0a36cc1205ad62ac6b13e52f6a0f8a620bf9698cbeeb5
     370057dfc5df742fc7b9fe6839bf2dd4bb
```

D.4 phpseclib v3.0 (relaxed mode)

D.4.1 Leniency in checking parameter field (CVE-2021-30130) [SF#1]. The implementation in the relaxed mode, uses the parsing-based (*i.e.*, decoding-based) approach to extract hash value and hash function from the encoded message. It then applies the hash function (whose ID is extracted from the encoded message) to the received message in order to obtain hash value, which is then compared with the extracted hash value from the encoded message. More specifically, it applies `decodeBER()` on the ASN.1 portion of the encoded message to obtain TLVs in the form of its internal structure (called `$decoded`). It then calls `asn1map()` function to extract `DigestInfo` (*i.e.*, hash function OID and parameter (`digestAlgorithm`) as well as the hash value (`digest`)). This is done by providing a blueprint specifying how the TLVs are supposed be nested given their types (*i.e.*, their identifier octets). More specifically, the blueprint mapping specifies, we should have a sequence TLV, that has two children called `digestAlgorithm` and `digest`, where `digest` has `octet string` type (`0x04` as identifier octet), and `digestAlgorithm` is a sequence TLV having two children. First child is algorithm with object identifier type (`0x06`) and second child is called parameters whose type is specified as `any` (but not `NULL` type). This leniency, at the very least, can cause interoperability issue. The implementation mistakenly accepts a signature whose encoded message has any type (than `NULL` type) in its parameter TLV section. But the implication can get much worse because the parameter value (*i.e.*, the content octets of the parameter section) are not being checked to actually match the `NULL` type. That is, the implementation does not verify that: *(i)* the parameter value should be absent; and *(ii)* the length octet is `0x00`. Instead, as `asn1map()` function is recursively called to get to the parameter section, it first decodes the type to understand what this type is, because according to the blueprint mapping it can be `any` type. Once decoded, if it is `NULL` type, the code just returns empty string, ignoring the actual content octets. But the parameter type does not even have to be `NULL` type as mentioned before, and some other types work as well because the `asn1map()` callee accepts any type for the parameter type. Considering that length octet here uses the definite short form, we are able to inject `79` random bytes as parameter value and get validated, given our settings described in section B. Since the relaxed version accepts BER, and as we know in BER, indefinite form is also allowed for the length octet of the top ASN.1 sequence, we are able to stack more garbage bytes into the parameter value (i.e., content octets) and extend the unchecked area from `79` bytes to `114` bytes out of `256` total bytes.

By performing root cause analysis, we can spot in line `5` in the snippet below from `asn1map()` function in `phpseclib/File/ASN1.php` that `any` instead of NULL type is used in the mapping blueprint.

```
1  public static function asn1map($decoded, $mapping,
       $special = [])
2  {
3    ...
4    switch (true) {
5      case $mapping['type'] == self::TYPE_ANY:
6        $intype = $decoded['type'];
7        ...
8        $inmap = self::ANY_MAP[$intype];
9        if (is_string($inmap)) {
10         return [$inmap => self::asn1map($decoded, ['
               type' => $intype] + $mapping, $special)
               ];
11       }
12       break;
13   ...
```

Also, as shown below in `asn1map()` function, when decoded type is recognized as `NULL`, then the content octets are not even checked here to be absent nor are passed to be checked by callee.

```
...
case self::TYPE_OCTET_STRING:
    return $decoded['content'];
case self::TYPE_NULL:
    return '';
case self::TYPE_BOOLEAN:
    return $decoded['content'];
...
```

Here is another snippet taken from the function `rsassa_pkcs1_v1_5_relaxed_verify()` in `phpseclib/Crypt/RSA/PublicKey.php`, where whatever is decoded as parameter is not being checked at all. The implementation only checks the hash algorithm and uses that to verify the signature.

```
$decoded = ASN1::asn1map($decoded[0], DigestInfo::
     MAP);
if (!isset($decoded) || $decoded === false) {
    return false;
}

if (!isset($oids[$decoded['digestAlgorithm']['
     algorithm']])) {
    return false;
}

$hash = $decoded['digestAlgorithm']['algorithm'];
$hash = substr($hash, 0, 3) == 'id-' ?
    substr($hash, 3) :
    $hash;
$hash = new Hash($hash);
$em = $hash->hash($m);
$em2 = $decoded['digest'];

return hash_equals($em, $em2);
```

Reproducing this bug can be done using the below concrete values, found by Morpheus, given the parameter settings provided in section B.

Example: 114 garbage bytes are added by misusing indefinite form of length octet in BER and value of parameter TLV being not checked:

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00
     3080307f0609608648016503040201057288888888888888888888888888888
     8888888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888804207509e5bda0c762d2bac7f90d758b5b2263
     fa01ccbc542ab5e3df163be08e6ca90000
S  = 0xac9e1f97acdbcd247ea79013d07e2f9ffe3da45f42cbf43c1a425524c6
     e4fe7242149cd43cbcf7c23cb64786e3b8dbcb5652ecf5009836f01f07ed
     d10c0a437291728d76049b17a624151c102efe304cae1c27939a856f9a1b
     14f296ff75702a6a53f47e52d46354966ba083fbb749a07b2b14509ab4c5
     22f6b3e48f9c78ab467a2bbfdf16cdcdd7bba3045ee7c4413e175b712f39
     5a8ae4dde7af82062facf214aa4f830e38a4f0aa8cfa2c3dcf74a228a7dc
     10b6ffb2ea3bcc4ea3a8a3c597c3be80fe2ba0bc0334693b834c03b8bc83
     4009069f53d2641f953b4aab00b14baf873cbbb95faa1f2f18197896b6ef
     8df3d413ea08e2b3fb31d7dba7eb5c4de8
```

D.4.2 Leniency in decoding hash function ID's content octets [ML#1]. To encode Object Identifier to an ASN.1 DER/BER encoded string, there are some specific instructions to follow [10]. But to the extent related to this vulnerability, any octet value less than 128 will be encoded as is in one byte, while any octet greater than or equal 128 (in decimal) will be encoded using transitional byte (or bytes) and remaining value byte. For example, in SHA-256 OID, which is `2.16.840.1.101.3.4.2.1`, the forth octet, 1, will be encoded as `0x01`, while the third octet, `840`, will be encoded as `0x8648`, where `0x86` is the transitional byte and `0x48` is the remaining value byte. During the decode operation, the reverse needs to be done to extract the OID. However, the phpseclib v3.0 (relaxed mode) has a leniency in decoding an OID octet whose value greater than or equal 128.

Example: 79 garbage bytes (`0x88`s) are appended to a valid SHA-256 OID (we can even add more garbage bytes, 116 bytes, by misusing indefinite form of length octet in BER and removing parameter TLV):

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffff003080305c06586086480165030402018888888888888888888
     8888888888888888888888888888888888888888888888888888888888888888
     8888888888888888888888888888888888888888888888888888888888888888
     88888888888888888888888888050004207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0x2d8c20ccaf93bce59e98623997f9c8e4b52cd5f2f1f0541aea19c12e2b
     31ccd3ef7d606940c6e626ed713ce318f082c56a40088167a490b54ecf24
     6a8b0e4565afe8d8808587e402a6c607a4891efef0b29f0dbb534ce4c4a6
     52f06f6dd5814636aa252bac8aa94b0c53a4ac5eb28ec8384b7d60731ebb
     a132a898a738b02dae78f350985ea8ab5ecc8203b3d6a1c8f74d92ac979f
     f31e9dea34c1411fee86ebaef2960cf03a5c9d5030d0cde0e27940a14a5d
     9e9a85d4f3e6ecc97cbdfce8fa64fe8cb82fe9dc7a174f11249181b7145f
     d3e8befb63f3f860db2f827b403d43b00e893036c75b6aee7d8d8de47570
     7f5780315b1c44974bd5f8fb3350aec54c
```

D.4.3 Leniency in checking form bit of an identifier octet [ML#2]. Similar to wpa_supplicant and hostapd, the phpseclib v3.0 (relaxed mode) implementation has a leniency in handling an identifier octet by failing to check the form bit as discussed in Appendix D.2.3.

D.4.4 Incorrect extended tag decoding of identifier octets [ML#3]. Similar to wpa_supplicant and hostapd, the phpseclib v3.0 (relaxed mode) implementation suffers from a logical flaw in decoding of the extended tags for identifier octets, as discussed in Appendix D.2.4

D.5 jsrsasign (v10.1.13)

D.5.1 Leniency in checking the prefix bytes [ML#1]. jsrsasign v10.1.13 is lenient in checking the prefix bytes and some other invalid signatures are mistakenly recognized to be valid. As can be seen in the below snippet from `RSAKey.prototype.verify()` function in `jsrsasign.js` file, the initial `0x00` bytes will be ignored when octet strings are converted to integer (line 4)) and converted back to octet strings after taking modular exponentiation to the power public exponent (line 8). Then the naive regex pattern being used in line 9 to strip the padding bytes only checks for the initial string `1f+00`, if exists any. That is, this pattern matching not only accepts the benign case where we have block type byte followed by the padding string (as in `0x01FF...FF00`) but also accepts the prefix string such as `0x01F...FF00` where block type byte is `0x01F`. Also, all padding bytes can be removed (as in `0x01F00`) or all can be of value `0x00` and still get validated by the signature verification.

```
1   RSAKey.prototype.verify = function (f, j) {
2     j = j.replace(_RE_HEXDECONLY, "");
3     j = j.replace(/[ \n]+/g, "");
4     var b = parseBigInt(j, 16);
5     if (b.bitLength() > this.n.bitLength()) {
6       return 0;
7     }
8     var i = this.doPublic(b);
9     var e = i.toString(16).replace(/^1f+00/, "");
10    var g =
          _rsasign_getAlgNameAndHashFromHexDisgestInfo(
          e);
11    if (g.length == 0) {
12      return false;
13    }
```

```
  var d = g[0];
  var h = g[1];
  var a = function (k) {
    return KJUR.crypto.Util.hashString(k, d);
  };
  var c = a(f);
  return h == c;
};
```

The below concrete values can be used to reproduce triggering this bug, given the parameter settings provided in section B.

Example#1: Padding bytes with length 1:

```
EM = 0x0001ff003031300d060960864801650304020105000420 7509e5bda0c7
     62d2bac7f90d758b5b2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0x3fa376d1e4f51e9ff32636fe4d5c1b22b9f3d2e5d017ea79168e8e2973
     16c4ca980827ad333f983bf9628e212a21f5fe5f2bde47088ef6b002cef3
     1f9ce395abd6400f2ddfc8c2d152174ff47bf9e396843033c752df861305
     4692ef0489fb59e3bd31e30ed18d83474c898236d4691b03d80c46c5fdbb
     1e8c5901c65748d0fc6857b64cd6baab722ac827d1d1e2cb1638afebe835
     47115747590eb83808e3dd99df942dc45461bb54efc5e52dc989af33c151
     8612571288de34542241b3ae98a3600499206be8c2fb56b14ca967712161
     666adabd8f79723d5b274c5cf8c84f5b2e158294662de15e897b298c9203
     829bc846eeccce9b424d6cf7a1d9fa284e
```

Example#2: All zero bytes as prefix:

```
EM = 0x0000000000000000000000000000000000000000000000000000000000
     000000000000000000000000000000000000000000000000000000000000
     000000000000000000000000000000000000000000000000000000000000
     000000000000000000000000000000000000000000000000000000000000
     000000000000000000000000000000000000000000000000000000000000
     000000000000000000000000000000000000000000000000000000000000
     00000000000000000000000000000000000000000000000000003031300d
     0609608648016503040201050004207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0xe5c9db5fb1f81cf431a568caeea54e3e7a3d54a2ae16929ecf52da9ea7
     018f18e7d3b2c3968a338d8b0cd0233c3073a25689da5310f52f2c14bf89
     c81ab2d8e2a0a5bef4403199aaa7a2bef6af6b33b15828f646358193c4ef
     8360c48e70f6e632b81432910b94b73854413fdb61f5377d88c5bb9848cd
     301b6e7609f90f4af94cbfae053fe7e9bcedd1c905d8a26f01133607d8f8
     1efe993ad9daf78188ddbe6496dd7b6bea2cb3617708b96ccd9e0ca2398d
     d34d81567d2644f7a3ae4956a674e389aab9d8f1e732759209ddf17fd01c
     c347e6c74464c03e4d57be3052582f04c8553a6ebc6046742c659d7f542b
     4c72f8966dfde06036ad47d190dcddd03a
```

Infeasibility of signature forgery attack. As discussed above jsrsasign is willing to take a prefix and padding of all zeros. As such, the remaining chunk (let us call it R) of ASN.1 DER structure (including the hash value) would be 406 bit long, assuming the parameter settings provided in section B. In other words, $R < 2^{406}$. The distance between two perfect cubes of this size is less than 2^{272}. In that case, assuming that the output of `SHA-256` hash function is uniformly at random, the probability of hitting a perfect cube under a chosen message is 2^{-272}; at that cost it might be easier to just factorize the 2048-bit modulus.

In fact, given the content of R that we need to match for the `SHA-256` case, the distance between two perfect cubes might even be larger than the size of the hash itself (256 bit).

D.6 CryptX (v0.070) & LibTomCrypt (v1.18.2)

CryptX is a Perl module providing a cryptography based on LibTomCrypt library. Therefore, they suffer from the same minor leniency as explained below.

D.6.1 Leniency in checking form bit of an identifier octet [ML#1].

Similar to wpa_supplicant and hostapd, the LibTomCrypt library (and thus CryptX) suffer from a flaw where the form bit of some identifier octets are not being checked which can cause interoperability issue. This issue affects `octet string` type used in hash value encoding structure and the `object identifier` type of hash ID encoding structure.

Example#1: Incorrect encoding of `octet string` type (`0x24` instead of the correct encoding `0x04`) used in the hash value structure is accepted as a valid signature's encoded message by the implementations.

```
EM = 0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffff003031300d06
     09608648016503040201050024207509e5bda0c762d2bac7f90d758b5b22
     63fa01ccbc542ab5e3df163be08e6ca9
S  = 1ee08947536e6b11d8923c3b00061d26a6933b5345077ea0214fdcbcc1ad
     68395008ff709117047e6b01dd2a371dfa032c0732abc86ab2e0273bbd0d
     fe6b1c769e21bb9079982801d8f72e01be3244959312ab09bb8f88572dc2
     3216719b9810c73edf826749604feb8da1345f83f0209271aca462c1235b
     4cb4ba538f85a9c03dd1dde1856fe73fd86b95566df2dfe8b0895c34489b
     97e02c8e48dabad7067619edec6267a776fa416fbcac0fcacf3efa7852ce
     33ed63a9149c685c303d98c3dc37ee87521bc5b130377345fc95c87aa485
     05470deaf6fb1064df041e3f03322b1ec90d3608deb17bf77f47066ecc6c
     511bfba69eed6da42881dcce603fcb2a
```

Example#2: Incorrect encoding of `object identifier` type (`0x26` instead of the correct encoding `0x06`) used in the hash function OID structure is accepted as a valid signature's encoded message by the implementations.

```
EM = 0x0001ffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     ffffffffffffffffffffffffffffffffffffffffffffffffff003031300d
     2609608648016503040201050004207509e5bda0c762d2bac7f90d758b5b
     2263fa01ccbc542ab5e3df163be08e6ca9
S  = 0x8df69d774c6ac8b5f8aa16576ca37a4f948706c5daecb3c15cfd247a76
     57616b2bbb786b50158cac8c23e3289d300d3fbb82380b8746d929df36bd
     af43a5fc5d1d04c61c98d47c22de02d051be3ba9e42b1c47aa519266d4ca
     e244e5ce99b24771a13a7c8c7b08868a3eccf70b4bc7570d5131a1ac8943
     d91b0151c39da2ad75cd1b9a697d100eef6747217df581b272cfd1f549a9
     01ff4951036a4eb28fd2ea1e9df3fa9fa457663f4259be8e5f2f2fb84f83
     1a0ca5320e2b79f04a17830f43062c4c8fc0d0b1ff90567f3342d524f682
     ca26661caadf4272f2585e6013a92bfa68de72fe6174096890e4296aedd7
     2da43aa508007df53fb852bd7162ab635b
```