Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification

Sze Yiu Chau*, Moosa Yahyazadeh†, Omar Chowdhury†, Aniket Kate*, Ninghui Li*,
{schau,aniket,ninghui}@purdue.edu, Purdue University*
{moosa-yahyazadeh,omar-chowdhury}@uiowa.edu, The University of Iowa†

***Abstract*— We discuss how symbolic execution can be used to not only find low-level errors but also analyze the semantic correctness of protocol implementations. To avoid manually crafting test cases, we propose a strategy of meta-level search, which leverages constraints stemmed from the input formats to automatically generate concolic test cases. Additionally, to aid root-cause analysis, we develop constraint provenance tracking (CPT), a mechanism that associates atomic sub-formulas of path constraints with their corresponding source level origins. We demonstrate the power of symbolic analysis with a case study on PKCS#1 v1.5 signature verification. Leveraging meta-level search and CPT, we analyzed 15 recent open-source implementations using symbolic execution and found semantic flaws in 6 of them. Further analysis of these flaws showed that 4 implementations are susceptible to new variants of the Bleichenbacher low-exponent RSA signature forgery. One implementation suffers from potential denial of service attacks with purposefully crafted signatures. All our findings have been responsibly shared with the affected vendors. Among the flaws discovered, 6 new CVEs have been assigned to the immediately exploitable ones.**

I. Introduction

Developing a deployable cryptographic protocol is by no means an easy feat. The journey from theory to practice is often long and arduous, and a small misstep can have the security guarantees that are backed by years of thorough analysis completely undone. Given well-defined cryptographic constructs originated from mathematical problems that are believed to be hard to solve, proving their functional correctness with respect to the relevant assumptions and security models is hardly the end of the journey. Because of the restrictive assumptions used in designing cryptographic constructs, in reality, additional glue protocols are often needed to generalize such constructs into being able to handle inputs of diverse length and formats. Sometimes glue protocols are also used to wrap around cryptographic constructs for exploiting the duality of certain security guarantees to achieve alternative properties. After careful designs have been devised and standardized, it is also necessary for implementations to faithfully adhere to the specification, in order to ensure the retention of the original designed security and functionality goals in actual deployments. Implementations that deviate from the standard and do not achieve the prescribed level of robustness can lead to a plethora of attacks [9], [20], [22], [27].

The PKCS#1 v1.5 signature scheme, surrounding the RSA algorithm, is one such glue protocol that is widely deployed in practice. Used in popular secure communication protocols like SSL/TLS and SSH, it has also been adapted for other scenarios like signing software. Prior work has demonstrated that lenient implementations of PKCS#1 v1.5 signature verification can be exploited in specific settings (*e.g.*, when small public exponents are being used) to allow the forgery of digital signatures without possession of the private exponent nor factorizing the modulus [3], [5], [13], [20], [24], [25], [27]. The identification of such implementation flaws, however, has been mostly based on manual code inspection [20], [27]. The focus of this paper is thus to develop a systematic and highly automated approach for analyzing *semantic correctness* of implementations of protocols like PKCS#1 v1.5 signature verification, that is, whether the code adheres to and enforces what the specification prescribes.

***Our approach*.** For identifying semantic weaknesses of protocol implementations, we propose to perform symbolic analysis of the software [26]. Directly applying off-the-shelf symbolic execution tools [14], [28] to test PKCS#1 v1.5 implementations, however, suffers from scalability challenges. This is due to the fact that the inputs to such protocols are often structured with variable length fields (*e.g.*, padding), and can sometimes contain sophisticated ASN.1 objects (*e.g.*, metadata).

One might question the applicability of symbolic analysis on implementations of a cryptographic protocol. The key intuition that we leverage in our approach, is that while the underlying mathematics of cryptographic constructs are typically non-linear in nature, which are often difficult to analyze with constraint solvers, the various variable-sized components used in glue protocols like PKCS#1 v1.5 often exhibit linear relations among themselves and with the input buffer (*e.g.*, sum of component lengths should equal to a certain expected value). Using linear constraints stemming from such relations, we can guide symbolic execution into automatically generating many meaningful concolic test cases, a technique we refer to as *meta-level search*.

To further address scalability challenges faced by symbolic execution, we draw insights from the so-called human-in-the-loop idea [21]. With domain knowledge on the protocol design and input formats, human expertise can partition the input space in a coarse-grained fashion by grouping together parts

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23xxx
www.ndss-symposium.org

of the input buffer that should be analyzed simultaneously, making them symbolic while leaving the rest concrete. A good partition strategy should constrain and guide symbolic execution to focus on subproblems that are much easier to efficiently and exhaustively search than the original unconstrained input space, and hence achieve good coverage while avoiding intractable path explosions due to loops and recursions.

To facilitate root-cause analysis of an identified deviation, we design and develop a *constraint provenance tracking* (CPT) mechanism that maps the different clauses of each path constraint generated by symbolic execution to their source level origin, which can be used to understand where certain decisions were being made inside the source tree. Our carefully designed CPT mechanism has been demonstrated to incur only modest overhead while maintaining sufficient information for identifying the root-cause of deviations in the source code.

Case Study. The PKCS#1 v1.5 signature scheme is a good candidate for demonstrating the effectiveness of our approach in analyzing semantic correctness, as the protocol itself involves diverse glue components. As we will explain later, to our surprise, even after a decade since the discovery of the original vulnerability [20], several implementations still fail to faithfully and robustly implement the prescribed verification logic, resulting in new variants of the reported attack.

Findings. To show the efficacy of our approach, we first use it to analyze 2 legacy implementations of PKCS#1 v1.5 signature verification that are known to be vulnerable. Our analysis identified not only the known exploitable flaws, but also revealed some additional weaknesses. We then analyze 15 recent open-source implementations with our approach. Our analysis revealed that 6 of these implementations (i.e., strongSwan 5.6.3, Openswan 2.6.50, axTLS 2.1.3, mbedTLS 2.4.2, MatrixSSL 3.9.1, and libtomcrypt 1.16) exhibit various semantic correctness issues in their signature verification logic. Our analysis in an existing theoretical framework shows that 4 of these weak implementations are in fact susceptible to novel variants of Bleichenbacher's low-exponent RSA signature forgery attack [20], [27], due to some new forms of weaknesses unreported before. Exploiting these newly found weaknesses, forging a digital signature does not require the adversary to carry out many brute-force trials as described in previous work [27]. Contrary to common wisdom, in some cases, choosing a larger security parameter (*i.e.*, modulus) actually makes various attacks easier to succeed, and there are still key generation programs that mandate small public exponents [7]. One particular denial of service attack against axTLS 2.1.3 exploiting its signature verification weakness can be launched even if no Certificate Authorities use small public exponents. Among the numerous weaknesses discovered, 6 new CVEs have been assigned to the exploitable ones.

Contributions. This paper makes the following contributions:

1) We propose and develop a principled and practical approach based on symbolic execution that enables the identification of exploitable flaws in implementations of PKCS#1 v1.5 signature verification. Specifically, we discuss how to enhance symbolic execution with *meta-level search* in Section II.
2) To aid root-cause analysis when analyzing semantic correctness with symbolic execution, we design and implement a constraint provenance tracker; which is of independent interest. We explain in Section III how this can help identify root causes of observed implementation deviations with only a modest overhead.
3) We demonstrate our approach with a case study on implementations of PKCS#1 v1.5 signature verification. Our analysis of 2 known buggy (Section IV-D) and 15 recent implementations (Section V) of PKCS#1 v1.5 not only led to the discovery of known vulnerabilities but also various new forms of weaknesses. We also provide theoretical analysis and proof-of-concept attacks based on our new findings in Section VI.

II. Symbolic Execution with Meta-level Search

While symbolic execution is a time-tested means for analyzing programs, the practicality challenges that it faces are also well understood. When dealing with complex structured inputs, one strategy to workaround scalability issues is to draw on domain knowledge to strategically mix concrete values with symbolic variables in the (concolic) test input. When done correctly, this should allow symbolic execution to reach beyond the input parsing code (which makes frequent use of loops and recursions) and explore the post-parsing decision making logic.

As explained in previous work [15], inputs like X.509 certificates that are DER-encoded ASN.1 objects, can be viewed as a tree of $\{$*Tag, Length, Value*$\}$ triplets, where the length of *Value* bytes is explicitly given. Hence, if all the *Tag* and *Length* are fixed to concrete values, the positions of where *Value* begins and ends in a test input buffer would also be fixed. Hence, one can generate a few concrete inputs, and manually mark *Value* bytes of interests as symbolic to obtain meaningful concolic test cases. In fact, just a handful of such manually produced test cases managed to uncover a variety of verification problems [15].

However, cryptographic glue protocols like PKCS#1 v1.5 signatures sometimes involve not only an encoded ASN.1 object, but also input components used for padding purposes, where the length is often implicitly given by an explicit termination indicator. In PKCS#1 v1.5, since padding comes before its ASN.1 structure, the extra room gained due to (incorrectly) short padding can be hidden in any later parts of the input buffer, including many leaf nodes of the encoded ASN.1 object. This means there could be many combinations of lengths of components that constitute the input buffer, all meaningful for testing. Consequently, the concretization strategy used in previous work [15] in this case requires a huge amount of manual effort to enumerate and prepare concolic inputs, and would easily miss out on meaningful corner cases.

To achieve a high degree of automation while preserving a good test coverage, we propose to use symbolic variables not only as test inputs, but also to capture some high-level abstractions of how different portions of the test inputs could be mutated, and let the SMT solver decide whether such mutations are possible during symbolic execution. The key insight is that, the lengths of input components used by protocols like PKCS#1 v1.5 exhibit linear relations with each other. For example, the size of padding and all the other components together should be exactly the size of the modulus, and in benign cases, the length of a parent node in an encoded

ASN.1 object is given by the sum of the size of all its child nodes. By programatically describing such constraints, symbolic execution can *automatically* explore combinations of possible component lengths, and generate concolic test cases on the fly by mutating and packing components according to satisfiable constraints.

Given that the input formats of many other protocols also exhibit similar patterns, the *meta-level search* technique should be applicable to them as well. We will explain how to fit this technique specifically for PKCS#1 v1.5 signatures and discuss other engineering details in Section IV.

III. Constraint Provenance Tracking for Easier Root Cause Analysis

In this section, we present the design, implementation, and empirical evaluation of the constraint provenance tracking (CPT) mechanism. CPT aids one to identify the underlying root-cause of an implementation deviation, identified through the analysis of the relevant path constraints generated by symbolic execution. CPT is of independent interest in the context of semantic correctness checking, as it can be used for many other protocols beyond PKCS#1 v1.5.

A. Motivation

While the logical formulas extracted by symbolic execution capture the implemented decision-making logic of the test target with respect to its inputs, which enable analysis of semantic correctness and provide a common-ground for differential testing as demonstrated by previous work [15], we argue that after discrepancies have been identified, a root-cause analysis from formula level back to code level is non-trivial to perform, as multiple different code locations of an implementation could have contributed to the various constraints being imposed on a specific symbolic variable. This is further exacerbated by the fact that, modern symbolic engines, like KLEE for example, would actively simplify and rewrite path constraints in order to reduce the time spent on constraint solving [14].

Take the following code snippet as a running example. Assuming that each `char` is 1-byte long and `A` is a symbolic variable, a symbolic execution engine like KLEE would discover 3 possible execution paths, with the return value being 0, 1, and 2, respectively.

```
int foo( char A ){
  char b = 10, c = 11;
  if (!memcmp(&A, &c, 1))
    return 0;
  if (memcmp(&A, &b, 1))
    return 1;
  return 2;
}
```

```
(Eq 10 (Read w8 0 A))
```

Example 1: A code snippet with 3 execution paths. The path constraint shown above corresponds to the path that gives a return value of 2.

Although the path that returns 2 *falsifies* the two branching conditions due to the `if` statements (i.e., $\mathtt{A}=11$ and $\mathtt{A}\neq10$), in the end, the simplified constraint only contains the falsification of the second branching condition (i.e., $\mathtt{A} \neq 10$), as shown in the path constraint. This is because the falsification of the second `if` condition imposes a more specific constraint on the symbolic variable than the first one, and a simplification of the path constraints would discard the inexact clauses in favor of keeping only the more specific and restrictive ones (i.e., $\mathtt{A} \neq 11 \wedge \mathtt{A} = 10 \leftrightarrow \mathtt{A} = 10$).

As illustrated by the example above, although the extracted path constraints faithfully capture the implemented logic, using them to trace where decisions were made inside the code is not necessarily straightforward even on a toy example.

In order to make root-cause analysis easier when it comes to finding bugs with symbolic execution, on top of merely harvesting the final optimized path constraints like previous work did [15], we propose a new feature to be added to the execution engine, dubbed *Constraint Provenance Tracking* (CPT). The main idea is that, during symbolic execution, when a new clause is to be introduced, the engine can associate some source level origin (*e.g.*, file name and line number) with the newly added clause, and export them upon completion of the execution. We envision that when it comes to finding root-causes of implementation flaws, this is better than stepping through an execution using a common debugger with a concrete input generated by the symbolic execution. This is because path constraints offer an abstraction at the level of symbolic variables, not program variables. While one might have to mentally keep track of potentially many different program variables and their algebraic relations when stepping with a debugger (especially when entering some generic functions, *e.g.*, a parser), in symbolic execution those are all resolved into constraints imposed on symbolic variables, and CPT offers insights on where did such impositions happen.

B. Design of CPT

*1) **Performance Considerations**:* While clause origin can be obtained directly from the debugging information produced by compilers, the constraint optimization needs to be handled delicately. On one hand, such optimizations significantly improve the runtime of symbolic execution [14], on the other, they are often irreversible, hindering root-cause analysis. Striving to balance both performance and usability, in our implementation of CPT, we introduce a separate container for path constraints and their source level origins. The intuition behind introducing the separate container is to let the engine continue performing optimization on the path constraints that drive the symbolic execution, so that runtime performance would not suffer significantly, but then the unoptimized clauses and their origins could be used to assist root-clause analysis. This is essentially trading space for time, and as we show later, the memory overhead is modest. We refer to this as CPT v1.0.

*2) **Function Filtering**:* Another interesting consideration in implementing CPT is what constitutes the origin of a clause. Blindly copying source level information corresponding to the current program counter during symbolic execution is possible, but many times this does not result in a meaningful outcome, because most real software systems are designed and implemented in a modular manner using various libraries.

Consider again the path that returns 2 from the running example (i.e., $\mathtt{A} \neq 11 \wedge \mathtt{A} = 10$), CPT v1.0 would give the following provenance information, where the origins of the clauses are shown to be from the instrumented C standard library which implements the `memcmp()` function:

```
(Eq false
   (Eq 11 (Read w8 0 A)))   @libc/string/memcmp.c:35
(Eq 10 (Read w8 0 A))       @libc/string/memcmp.c:35
```

While this is technically accurate, from the perspective of analyzing the semantic correctness of a protocol implementation, this is not particularly meaningful. In such a setting, one would most likely not be very interested in analyzing the implementation of the underlying low-level library (*e.g.* the C standard library) and would prefer to have instead the caller of `memcmp()` to be considered as the origin of the clauses.

To this end, we propose to trace stack frames and filter out functions that one would like to ignore in tracking origins of clauses. One can, for example, configure the CPT to not dive into functions from the C standard library through blacklisting exported functions known from the API, and track instead the caller of those functions as the clause origins, which would produce the following CPT output for the same path that returns 2, clearly more useful in understanding the semantics of a protocol implementation:

```
(Eq false
   (Eq 11 (Read w8 0 A)))  @Example1.c:3
(Eq 10 (Read w8 0 A))      @Example1.c:5
```

In addition to the C standard library, we have observed that several cryptography implementations have their own shim layers mimicking the standard library functions (*e.g.* `OPENSSL_memcmp()` in OpenSSL). This is often done for the sake of platform portability (*e.g.* use the C standard library and some platform-specific extensions if they are available, and use a custom imitation if they are not), and is sometimes used to provide custom constant-time alternatives to avoid timing side-channel leakages. All these additional functions can be filtered similarly in CPT as well.

We note that when filtering function calls, there are two possible heuristics. (1) One is to consider the most recent caller of the blacklisted library functions as the clause origin. (2) Another alternative is to consider function calls to have a boundary, where once a blacklisted function has been called, the execution stays in a blacklisted territory until that function returns. While the first heuristic is better at handling callback functions, *we have chosen heuristic 2*, because fully blacklisting all the library functions that CPT should not dive into (or, equivalently, whitelisting all the possible origin functions from a protocol implementation) could be complicated. For example, specific implementations of C standard libraries may use their own undocumented internal functions to implement functions that are exported in the API. Acquiring this knowledge ahead of time could be laborious and hinders generalization.

We use CPT v2.0 to refer to the CPT with function filtering heuristic 2. In the end, we implemented CPT v2.0 by adding less than 750 lines of code to the KLEE toolchain. We chose KLEE as our symbolic execution engine because it is widely used, robust, and is actively maintained.

C. Performance Evaluation

We now evaluate the performance of KLEE [14] equipped with CPT, and compare it with vanilla KLEE. The goal of this evaluation is to demonstrate that both the memory and runtime overheads induced by the CPT feature are tolerable, as a significant increase in either of the two would severely hinder the practicality of using KLEE in software testing.

TABLE I. PERFORMANCE EVALUATION OF KLEE WITH CPT (AVERAGE OVER 3 TRIALS)

Program	KLEE version	Paths Completed	Time (s)	maxMem (MB)	avgMem (MB)
[	*Original*	1789	63.06	29.75	27.01
	CPT v2.0	1789	62.76	29.79	27.07
base64	*Original*	2097957	3,600.01	41.42	34.57
	CPT v2.0	2091665	3,600.01	41.44	34.65
basename	*Original*	14070	9.20	22.81	22.59
	CPT v2.0	14070	9.15	22.88	22.64
cat	*Original*	2261170.67	3,600.01	23.68	23.24
	CPT v2.0	2248991.67	3,600.01	23.76	23.32
chcon	*Original*	480351	3,600.02	59.75	57.82
	CPT v2.0	477896	3,600.01	59.95	57.94
chgrp	*Original*	705117.33	3,600.04	479.42	286.97
	CPT v2.0	703403.33	3,600.05	478.46	288.03
chmod	*Original*	430347.00	3,600.05	393.30	221.09
	CPT v2.0	427392.33	3,600.12	378.08	211.71
chown	*Original*	550473.67	3,600.06	353.46	201.40
	CPT v2.0	543620.67	3,600.04	349.93	200.03
chroot	*Original*	1496	7.25	23.83	23.16
	CPT v2.0	1496	7.58	23.98	23.23
cksum	*Original*	2552	7.91	24.81	23.64
	CPT v2.0	2552	7.74	24.85	23.75
comm	*Original*	3895174.33	3,600.01	92.59	68.91
	CPT v2.0	3857897.33	3,600.01	91.86	68.51
cp	*Original*	497.00	3,625.05	28.76	28.37
	CPT v2.0	496.33	3,615.35	28.81	28.44
cut	*Original*	3824504.33	3,600.01	26.21	25.76
	CPT v2.0	3826345	3,600.01	26.31	25.84
date	*Original*	3564.67	3,602.71	60.79	39.59
	CPT v2.0	3560.00	3,602.82	61.21	39.60
dd	*Original*	1069290.67	3,600.02	26.48	26.07
	CPT v2.0	1075813.33	3,600.02	26.67	26.19
df	*Original*	5016.33	3,600.13	71.27	48.18
	CPT v2.0	4127.33	3,600.10	68.98	47.64
dircolors	*Original*	1019074.33	3,600.02	25.00	24.64
	CPT v2.0	1007623.67	3,600.03	25.21	24.77
dirname	*Original*	4167	8.42	23.43	22.93
	CPT v2.0	4167	8.52	23.53	22.98
du	*Original*	179.67	3,600.55	55.21	43.97
	CPT v2.0	179.33	3600.52	55.32	43.65
echo	*Original*	5134030	3,600.01	22.60	22.42
	CPT v2.0	5081012	3,600.01	22.91	22.54
env	*Original*	508649	942.72	24.04	23.07
	CPT v2.0	508649	925.12	24.13	23.15
expand	*Original*	3466952.00	3,600.01	63.96	54.49
	CPT v2.0	3377675.67	3,600.01	64.05	54.36
expr	*Original*	4653	3,600.09	363.95	212.91
	CPT v2.0	4645	3,600.14	363.82	212.58
factor	*Original*	618634.33	3,600.06	381.06	208.15
	CPT v2.0	619403.33	3,600.06	381.57	208.66
false	*Original*	23	0.08	20.85	20.68
	CPT v2.0	23	0.08	20.90	20.72
fmt	*Original*	1330	3,610.77	25.88	25.36
	CPT v2.0	1308	3,610.72	25.96	25.42
fold	*Original*	4498176	3,600.01	23.46	23.20
	CPT v2.0	4426844	3,600.01	23.56	23.28
head	*Original*	1445240	3,600.02	2,073.72	1,496.51
	CPT v2.0	1445458	3,600.03	2,073.82	1,497.35
hostid	*Original*	1022352.00	3,600.01	25.79	25.46
	CPT v2.0	1021386.67	3,600.04	25.99	25.59
hostname	*Original*	991770.67	3,600.01	25.23	24.89
	CPT v2.0	994186.67	3,600.01	25.40	25.01

The overheads are reported by measuring *time* and *memory* needed by KLEE (with and without CPT) to symbolically execute a suite of target programs. Following what had been previously investigated in the original KLEE paper [14], we use the GNU coreutils package[1] for our evaluation, which consists of various basic tools like `cat` and `ls` used on many Unix-like operating systems. Over the years, coreutils itself has been tested extensively, so we do not intend to find new bugs or achieve a higher code coverage in our experiments.

We follow the experiment setup [1] used in the KLEE paper [14] to run 2 different versions of KLEE on coreutils version 6.11 [2], that is, the original version of KLEE, and the one with CPT v2.0. For each version, we repeat the execution on each coreutil program 3 times and report the average values of runtime and memory measurements. The experiments were conducted on a machine powered by an Intel Core i7-6700 3.40GHz CPU and with 32GB RAM. Table I shows our

[1] https://www.gnu.org/software/coreutils/coreutils.html

measurements on the first 30 programs in coreutils.

To obtain measurement numbers in each experiment, we use the `klee-stat` tool provided by KLEE toolchain. For memory usage, we report both the peak (*maxMem*) and average consumption (*avgMem*), averaged over the 3 executions. Since some of the target programs need an enormous amount of time to finish, following previous work [1], [14], we halt an execution after 1 hour, which explains why some programs in Table I have a total runtime of about 3600 seconds (*e.g.*, `base64`, `cat`, and `chcon`). In such cases, the mere total execution time is insufficient in showing the time overhead. Hence we also report the average number of completed paths during the 3 executions, which can be used to compare the runtime efficiency of the different versions of KLEE.

To make the number of completed paths comparable, and since we are not focused on code coverage, we also changed the search heuristic used by KLEE into a depth-first search (DFS), instead of a random search as prescribed by the recipe [1], to avoid non-determinism. We also increased the maximum memory usage for each execution to 16GB from the prescribed 1GB [1]. However, as can be seen in Table I, none of the tested programs approached close to this limit.

All in all, the two versions of KLEE yielded comparable total runtime (or, paths completed) and memory usages. CPT v2.0 in general consumes a little more memory and is slightly slower than the original KLEE, though the overheads are insignificant. In the rest of the paper, unless explicitly mentioned, we will be using KLEE with CPT v2.0 by default.

IV. A Case Study on PKCS#1 v1.5 RSA Signature Verification

We center our analysis around the problem of PKCS#1 v1.5 signature verification. This is particularly suitable for showcasing the merit of enhancing symbolic execution with meta-level searching, as it features diverse glue components including explicitly terminated padding with implicit length, as well as a sophisticated ASN.1 structure. Despite the PKCS#1 family has newer algorithms like RSA-PSS [RFC8017], the v1.5 signature scheme continues to be widely-used in Web PKI and other security-critical network protocols like *SSH* [RFC4253] and *IKEv2* [RFC7296] for authentication purposes.

A. Technical Background

In this section, we provide a brief overview of RSA signature verification while using PKCS#1 v1.5 as the padding scheme. For the ease of exposition, we provide a list of the notations we use and their meaning in Table II.

Following the usual RSA notations, we use d, e, and n to denote the RSA private exponent, public exponent, and modulus, respectively. $\langle n, e\rangle$ constitutes an RSA public key. We use $|n|$ to denote the size of the modulus in bits. Suppose m is the message for which an RSA signature is to be generated. In the context of X.509 certificates (and CRLs), m would be the ASN.1 DER-encoded byte sequence of `tbsCertificate` (and `tbsCertList`) [RFC5280].

TABLE II. Notation Used

Symbol	Description	Symbol	Description
n	RSA modulus	e	RSA Public Exponent
d	RSA Private Exponent	$\|n\|$	length of modulus in bits
m	message to be signed	m_v	message received by verifier
I	formatted input to the signer's RSA operation		
S	Signature, $S \equiv I^d \bmod n$ in benign cases		
O	verifier's RSA output, $O \equiv S^e \bmod n$		
$H(m_s)$	signer's version of $H(m)$, contained inside O		
$H(m_v)$	verifier's computed hash of m_v		
I_v	verifier's construction of I given $H(m_v)$		

Symbol	Description	Symbol	Description
`BT`	Block Type	`PB`	Padding Bytes
`AS`	ASN.1 Structure, containing $H(m_s)$		
w	ASN.1 Length of `AS.DigestInfo`		
u	ASN.1 Length of algorithm OID		
x	ASN.1 Length of `AlgorithmIdentifier`		
y	ASN.1 Length of parameters		
z	ASN.1 Length of `Digest`		

Benign signature generation. For generating an RSA signature of message m in accordance to PKCS#1 v1.5, the signer first computes the hash of m, denoted $H(m)$, based on the hash algorithm of choice (e.g., SHA-1). Then, $H(m)$ and the corresponding meta-data identifying the used hash algorithm and other relevant parameters (if any) are packed into an ASN.1 DER-encoded structure. The necessary amount of padding and other meta-data are prepended to the ASN.1 structure to create a structured input I of size $|n|$, which is then used as an input to the signer's RSA operation. The exact format of I is discussed below. Then, the signature will be $S = I^d \bmod n$.

Signature verification. Upon receiving a signed object (say an X.509 certificate), the verifier parses S from it and computes $O := S^e \bmod n$, where O represents the output of the verifier's RSA operation, formatted just like I in correct cases. Given m_v (say `tbsCertificate` of a received certificate), the verifier then computes $H(m_v)$ and compare it against the $H(m_s)$ contained in O. Like previous work has discussed [27], this comparison could be done in the following two manners.

Construction-based verification. Using this approach, the verifier takes $H(m_v)$ and prepares I_v, similar to how the signer is expected to prepare I prior to signing. If $I_v \equiv O$ then the signature is accepted.

Parsing-based verification. Many implementations seem to prefer a parsing-based approach, and this is where things can potentially go wrong. In essence, the goal of this approach is to parse $H(m_s)$ out of O. Many parsers are, however, too lenient even when O is malformed, which gives room for the so-called Bleichenbacher low-exponent brute-force attack.

Structured input (I) and output (O) format. In the benign case, I and O should be formatted as follows:

`0x00 || BT || PB || 0x00 || AS`

`BT` is often referred to as the block type [RFC2313], and `PB` represents the padding bytes. For the purpose of signature generation and verification, `BT` $\equiv$ `0x01` and `PB` $\equiv$ `0xFF 0xFF ... 0xFF`. Additionally, `PB` has to be at least 8-byte long, and also long enough such that there would be no extra bytes following `AS`. The `0x00` after `PB` signifies the end of padding. `AS` is an ASN.1 DER-encoded byte stream that looks like this (assuming $H()$ being SHA-1):

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                   // ASN.1 SEQUENCE, length = w
    /* [AlgorithmIdentifier] */
    30 x               // ASN.1 SEQUENCE, length = x
    06 u 2B 0E 03 02 1A    // ASN.1 OID, length = u
    05 y          // ASN.1 NULL parameter, length = y
/* [Digest] */
04 z              // ASN.1 OCTET STRING, length = z
  /* H(m), H()=SHA-1(), m = "hello world" */
  2A AE 6C 35 C9 4F CF B4 15 DB
  E9 5F 40 8B 9C E9 1E E8 46 ED
```

Since DER encoded ASN.1 structures are essentially a tree of $\{$*Tag, Length ,Value*$\}$ triplets, the length of a parent triplet is defined by the summation of the length of its child triplets. Assuming SHA-1, we can derive the following semantic relations among the different length variables for benign cases: $u = 5$; $z = 20$; $x = 2 + u + 2 + y$; $w = 2 + x + 2 + z$.

For most common hash algorithms like MD5, SHA-1, and the SHA-2 family, the algorithm parameter has to be `NULL` and $y \equiv 0$ [RFC2437, RFC4055]. Historically there were confusions on whether the `NULL` algorithm parameter can be omitted, but now both explicit `NULL` and absent parameters are considered to be legal and equivalent [RFC4055]. This could be a reason why some prefer parsing-based over construction-based, as in the latter approach the verifier would have to try at least two different constructions $\{I_{v_1},\ I_{v_2}\}$ to avoid falsely rejecting valid signatures. We focus on the explicit `NULL` parameter case in this paper, as it had been shown that the lenient processing of the parameter bytes can lead to signature forgery [27], and rejecting absent parameter is a compatibility issue easily identifiable with one concrete test case.

When PKCS#1 v1.5 signatures are used in other protocols like SSH and IKEv2 not involving X.509 certificates, the aforementioned steps work similarly with a different input message m (*e.g.*, m could be the transcript containing parameters that were exchanged during a key exchange algorithm).

B. Testing Deployed Implementations with Our Approach

We now discuss the different challenges and engineering details of how to make the implementations amenable to symbolic analysis. As discussed before, we use KLEE with CPT as our choice of symbolic analysis tool. Building an implementation for KLEE generally takes a few hours of trial-and-error to tune its build system into properly using LLVM.

1) ***Scalability Challenges****:* Since the length of O is given by $|n|$, for the best coverage and completeness, ideally one would test the verification code with a $\frac{|n|}{8}$-byte long symbolic buffer mimicking O. For implementations that use the parsing-based verification approach, however, since there are possibly many parsing loops and decisions depend on values of the input buffer, using one big symbolic buffer is not scalable.

To workaround scalability challenges, we use a two-stage solution. We first draw on domain knowledge to decompose the original problem into several smaller subproblems, each of which symbolic analysis can then efficiently and exhaustively search. Then for each subproblem we apply the meta-level search technique to automatically generate concolic test cases.

Stage 1. Coarse-grained decomposition of input space*.* In the first stage, we partition the input space influencing the exploration of the PKCS#1 v1.5 implementations in a coarse-grained fashion. Our coarse-grained partitioning resulted in three partitions, each corresponds to a dedicated test harness. For each implementation, the 3 test harnesses focus on testing various aspects of signature verification while avoiding scalability challenges. Across different implementations, each of the 3 test harnesses—denoted $\{$*TH1, TH2, TH3*$\}$—is focused on the same high-level aspect of testing. The test harnesses would invoke the implementations' PKCS#1 v1.5 signature verification functions, just like a normal application does. Depending on the API design of a specific implementation, the test harnesses also provide the appropriate verification parameters like an RSA public key, $H(m_v)$ (or in some cases, m_v directly) and a placeholder signature value.

Among the different harnesses, *TH1* is designed to investigate the checking of `BT`, `PB`, z the length of $H(m_s)$, and the algorithm parameters, while *TH2* is geared towards the matching of OID in `AlgorithmIdentifier`. Both *TH1* and *TH2* use a varying length of `PB` but the ASN.1 length variables u, w, x, y, z are kept concrete. In contrast, *TH3* has everything else concrete, reminiscent of a correct well-formed O, but u, w, x, y, z are made symbolic, to see how different length variables are being handled and whether an implementation would be tricked by absurd length values. In general, loops depending on unbounded symbolic variables poses threats to termination, however, as we would discuss below, in the context of PKCS#1 v1.5 signatures, one can assume all the length variables are bounded by some linear functions of $|n|$ and still achieve meaningful testing.

Stage 2. Meta-level search using relations between glue components*.* Following the meta-level search idea discussed in Section II, in both *TH1* and *TH2*, we provide linear constraints that describe the relations between $w, x, y, z, \frac{|n|}{8}$ and $|$`PB`$|$. As such, during symbolic execution, many different possible concolic test input buffers would be packed with respect to the given constraints in *TH1* and *TH2*, which effectively expand the two test harnesses automatically into many meaningful test cases, without the need to manually craft a large number of test harnesses, one for each test case. This is essentially a form *model counting*. Including effort of studying the PKCS#1 v1.5 specification, developing the meta-level search code for $\{$*TH1, TH2*$\}$ took a few days. This is however a one-time effort, as the code is generic and was reused across all implementations that we tested. Finally, *TH3* covers the extra cases where w, x, y, z are not constrained in terms of each other and $\frac{|n|}{8}$.

2) ***Memory Operations with Symbolic Sizes****:* We note, however, performing memory allocation and copy (*e.g.*, `malloc()` and `memcpy()`) with symbolic lengths would result in a concretization error where KLEE would try to concretize the length and continue the execution with one feasible concrete length value, hence missing out on some possible execution paths.

Explicit loop introduction*.* To avoid such concretization errors, when implementing the meta-level search in *TH1* and *TH2*, we use some simple counting `for`-loops, as shown below, to guide KLEE into trying different possible values of the symbolic lengths. What happens is that for each feasible value (with respect to known constraints that are imposed on those symbolic variables), KLEE would assign it to k and fork the execution before the memory allocation and copy, hence being able to try different lengths and not cutting through the search space due to concretization.

```
    size_t k; for (k = 0; i < sym_var; k++){}
```

```
/** execution forks with possible values of k **/
dest = malloc(k);      // k already concretized
memcpy(src, dest, k); // k already concretized
```

Bounding parameter length. Since explicit loop introduction is essentially trading time and space for coverage, it will not work practically if the range of possible values is very large. Fortunately, in PKCS#1 v1.5, the size of O is bounded by $|n|$. We leverage this observation to make our symbolic analysis practical, by focusing on a small $|n|$. Specifically, in our test harnesses, we assume the SHA-1 hash algorithm, as it is widely available in implementations, unlike some other older/newer hash functions, and that $|n|$ is 48-byte long (except for MatrixSSL, explained later), so that even after the minimum of 8-byte of `PB` there would still be at least 2 bytes that can be moved around during testing. Though in practice a 384-bit modulus is rarely used, and SHA-1 is now considered weak and under deprecation, since $|n|$ and the hash algorithm of choice are just parameters to the PKCS#1 v1.5 signature verification discussed in Section IV-A, assuming uniform implementations, our findings should be extensible to signatures made of a larger $|n|$ and other hash algorithms.

3) ***Accessing relevant functions for analysis:*** Finally, in order to make the implementation amenable to symbolic execution, one would need a customary, minuscule amount of modifications to the source tree. In this case, the modifications are made mainly to (1) change the visibility of certain internal functions; (2) inject the test buffer into the implementation's verification code. Test buffer injection is typically added to the underlying functions that implement the RSA public key operation which compute $O := S^e \bmod n$, easily identifiable with an initial test harness executed in an instrumented manner. Writing the test harnesses and adding the modifications generally take a few hours. In the case of unit tests (and stub functions) for signature verification are readily available (*e.g.* in Openswan), we can simply adapt and reuse their code.

C. Identifying semantic deviations

Path constraints extracted by symbolic execution can be analyzed in the following two ways to identify implementation flaws. When testing recent implementations, we would use both. Recall that PKCS#1 v1.5 is a deterministic padding scheme and we focus on the explicit `NULL` parameter case. For each test harness, if more than one accepting paths can be found by symbolic execution, then the implementation is highly likely to be deviant. (1) With CPT, one can inspect the path constraints and the origins of their clauses, as well as the generated test cases, to identify the faulty code. (2) To help highlight subtle weaknesses, we adopt the principle of differential testing [19] by *cross-validating* path constraints of different implementations, similar to previous work [15].

D. Feasibility Study

To validate the efficacy of our approach, we first apply it to test historic versions of OpenSSL and GnuTLS that are known to exhibit weaknesses in their signature verification, without using differential cross-validation for fairness reasons. The summary of results can be found in Table III.

As expected, both OpenSSL 0.9.7h and GnuTLS 1.4.2 use the parsing-based approach for verification. In fact, because both of them also perform some memory allocations based on parsed length variables that are made symbolic in *TH3*, they both needed explicit loop introduction as discussed before.

TABLE III. Result Summary of Testing Known Vulnerable PKCS#1 v1.5 Implementations with Symbolic Execution

Implementation (version)	**Test Harness**	**Lines Changed**	**Execution Time ‡**	**Total Path (Accepting)**
GnuTLS (1.4.2)	*TH1*	6	00:01:32	2073 (3)
	TH2		01:03:12	127608 (21)
	TH3	8	00:07:35	1582 (1)
OpenSSL (0.9.7h)	*TH1*	4	00:07:23	4008 (3)
	TH2		00:00:46	1432 (3)
	TH3	6	00:33:24	3005 (4)

‡ Execution Time measured on a commodity laptop with an Intel i7-3740QM CPU and 32GB DDR3 RAM running Ubuntu 16.04.

For OpenSSL 0.9.7h, the numerous accepting paths in *TH1, TH2* can be attributed to the fact that it accepts signatures containing trailing bytes after `AS`, which is exactly the original vulnerability that enables a signature forgery when $e = 3$ [4], [20]. On top of that, with *TH3*, we found that in addition to the one correct accepting path, there exists other erroneous ones. Specifically, we found that for the ASN.1 length variables y and z, besides the benign values of $y = 0$ and $z = 20$, it would also accept $y = 128$ and $z = 128$, which explains why there are four accepting paths. This is due to the leniency of the ASN.1 parser in OpenSSL 0.9.7h, which when given certain absurd length values, it would in some cases just use the actual number of remaining bytes as the length, yielding overly permissive acceptances during verification. Though not directly exploitable, this is nonetheless an interesting finding highlighting the power of symbolic analysis, and we are not aware of prior reports regarding this weakness.

For GnuTLS 1.4.2, the multiple accepting paths induced by *TH1* are due to the possibility of gaining extra free bytes with an incorrectly short padding and hiding them inside the algorithm parameter part of `AS`, which will then be ignored and not checked. This is the known flaw that enabled a low-exponent signature forgery [25], [27]. Additionally, with *TH3*, we found that there exist an opportunity to induce the parser into reading from illegal addresses, by giving u a special value. Specifically, assuming SHA-1, after the parser has reached but not consumed u, there are still 30 bytes remaining in `AS`. Despite the several sanity checks in place to make sure that the parsed length cannot be larger than what is remaining, by making u exactly 30, it does not violate the sanity checks, but at a later point when the parser attempts to read the actual OID value bytes, it would still be tricked into reading beyond `AS`, which resulted in a memory error caught by KLEE.

The 21 accepting paths (1 correct and 20 erroneous) induced by *TH2* in GnuTLS 1.4.2 can be attributed to how the parser leniently handles and accepts malformed algorithm OIDs. This over-permissiveness in signature verification does not seem to have been reported before.

By both recreating known vulnerabilities and finding new weaknesses in the old versions of GnuTLS and OpenSSL, we have demonstrated the efficacy of our proposed approach.

V. Findings on Recent Implementations

Here we present our findings of testing 15 recent open-source implementations of PKCS#1 v1.5 signature verification.

TABLE IV. Result Summary of Testing various New PKCS#1 v1.5 Implementations with Symbolic Execution

Implementation (version)	Test Harness	Lines Changed	Execution Time †	Total Path ‡ (Accepting)
axTLS (2.1.3)	*TH1*	7	01:42:14	1476 (6)
	TH2		00:00:05	21 (21)
	TH3	9	00:00:10	21 (1)
BearSSL (0.4)	*TH1*	3	00:01:55	3563 (1)
	TH2		00:00:06	42 (1)
	TH3		00:00:00	6 (1)
BoringSSL (3112)	*TH1*	3	00:06:09	3957 (1)
	TH2		00:00:08	26 (1)
	TH3		00:00:00	6 (1)
Dropbear SSH (2017.75)	*TH1*	4	00:46:10	1260 (1)
	TH2		00:00:11	23 (1)
	TH3		00:00:15	7 (1)
GnuTLS (3.5.12)	*TH1*	4	00:01:35	570 (1)
	TH2		00:00:06	22 (1)
	TH3		00:00:01	4 (1)
LibreSSL (2.5.4)	*TH1*	4	00:10:27	4008 (1)
	TH2		00:01:40	1151 (1)
	TH3	6	00:25:45	1802 (1)
libtomcrypt (1.16)	*TH1*	5	00:01:13	2262 (3)
	TH2	16	00:00:11	805 (3)
	TH3	5	00:04:49	7284 (1)
MatrixSSL (3.9.1) *Certificate*	*TH1*	8	00:01:54	4554 (1)
	TH2		00:00:04	202 (1)
	TH3		00:00:22	939 (2)
MatrixSSL (3.9.1) *CRL*	*TH1*	4	00:01:55	4574 (21)
	TH2		00:00:04	202 (61)
	TH3		00:00:07	350 (7)
mbedTLS (2.4.2)	*TH1*	7	00:14:56	51276 (1)
	TH2		00:00:03	26 (1)
	TH3		00:00:00	38 (1)
OpenSSH (7.7)	*TH1*	6	00:07:00	3768 (1)
	TH2		00:00:08	22 (1)
	TH3		00:00:00	2 (1)
OpenSSL (1.0.2l)	*TH1*	4	00:06:31	4008 (1)
	TH2		00:00:56	1148 (1)
	TH3	6	00:16:16	1673 (1)
Openswan (2.6.50) *	*TH1*	4	00:01:07	378 (1)
	TH2		00:00:04	26 (1)
	TH3		00:00:00	6 (1)
PuTTY (0.7)	*TH1*	12	00:03:22	3889 (1)
	TH2		00:00:07	42 (1)
	TH3		00:00:00	6 (1)
strongSwan (5.6.3) *	*TH1*	6	00:01:32	2262 (3)
	TH2		00:16:36	15747 (3)
	TH3		00:00:24	216 (6)
wolfSSL (3.11.0)	*TH1*	10	00:04:05	14316 (1)
	TH2		00:00:06	26 (1)
	TH3		00:00:00	6 (1)

† Execution Time measured on a commodity laptop with an Intel i7-3740QM CPU and 32GB DDR3 RAM running Ubuntu 16.04.
‡ Shaded cells indicate no discrepancies were found during cross-validation.
* Configured to use their own internal implementations of PKCS#1 v1.5.

We take the construction-based approach as the golden standard. For each of the test harnesses, while the occurrence of multiple accepting paths signifies problems, it is worth noting that just because an implementation gave only one accepting path does not mean that the implemented verification is robust and correct. In fact, as we show later, some lone accepting paths can still be overly permissive. The summary of results can be found in Table IV.

Cross-validation. For performing cross-validation, we use GnuTLS 3.5.12 as our anchor, as it seems to be using a robust construction-based signature verification, and it gave the smallest number of paths with *TH1*. We ran the cross-validation on a commodity laptop with at most 8 query instances in parallel at any time. For each implementation, cross-validating it against the anchor for a particular test harness typically finishes in the scale of minutes. In general, the exact time needed to solve such queries depends on the size and complexity of the constraints, but in this particular context, we have observed that the overall performance is around 1200 queries per every 10 seconds on our commodity laptop.

In the rest of this section, when we show code snippets, block comments with a single star are from the original source code, and those with double stars are our annotations.

*1) **Openswan 2.6.50**:* Openswan is a popular open source IPSec implementation, currently maintained by Xelerance Corporation. Depending on the target platform, Openswan can be configured to use NSS, or its own implementation based on GMP, for managing and processing public-key cryptography. We are particularly interested in testing the latter one.

The verification of PKCS#1 v1.5 RSA signatures in Openswan employs a hybrid approach. Given an O, everything before `AS` is processed by a parser, and then `AS` is checked against some known DER-encoded bytes and the expected $H(m_v)$, which explains why *TH2* and *TH3* both found only a small number of paths, similar to the other hybrid implementations like wolfSSL and BoringSSL. Those paths also successfully cross-validated against the anchor.

Interestingly, despite *TH1* yielding only 1 accepting path, Openswan turns out to have an exploitable vulnerability in its signature verification logic.

Ignoring padding bytes (CVE-2018-15836): As shown in Snippet 1, during verification, the parser calculates and enforces an expected length of padding. However, while the initial `0x00`, `BT`, and the end of padding `0x00` are verified, the actual padding is simply skipped over by the parser. Since the value of each padding byte is not being checked at all, for a signature verification to succeed, they can take arbitrarily any values. As we will explain later in Section VI-1, this simple but severe oversight can be exploited for a Bleichenbacher-style signature forgery.

Snippet 1. Padding Bytes skipped in Openswan 2.6.50

```
/* check signature contents */
/* verify padding (not including any DER digest info! */
padlen = sig_len - 3 - hash_len;
... ...
/* skip padding */
if(s[0] != 0x00 || s[1] != 0x01 || s[padlen+2] != 0x00)
  { return "3""SIG padding does not check out"; }
s += padlen + 3;
```

*2) **strongSwan 5.6.3**:* strongSwan is another popular open source IPSec implementation. Similar to Openswan, when it comes to public-key cryptography, strongSwan offers the choice of relying on other cryptographic libraries (*e.g.*, OpenSSL and libgcrypt), or using its own internal implementation, which happens to be also based on GMP. We are focused on testing the latter one. To our surprise, the strongSwan internal implementation of PKCS#1 v1.5 signature verification contains several weaknesses, many of which could be exploited for signature forgery.

Not checking algorithm parameter (CVE-2018-16152): *TH1* revealed that the strongSwan implementation does not reject O with extra garbage bytes hidden in the algorithm parameter, a classical flaw previously also found in other libraries [13], [25]. As such, a practical low-exponent signature forgery exploiting those unchecked bytes is possible [27].

Accepting trailing bytes after OID (CVE-2018-16151): *TH2* revealed another exploitable leniency exerted by the parser used by strongSwan during its signature verification. The `asn1_known_oid()` function is used to match a series of parsed OID encoded bytes against known OIDs, but the matching logic is implemented in a way that as soon as a known OID is found to match the prefix of the parsed bytes, it considers the match a success and does not care whether there are remaining bytes in the parsed OID left unconsumed. One can hence hide extra bytes after a correctly encoded OID, and as we will explain in Section VI-3, this can be exploited for a low-exponent signature forgery.

Accepting less than 8 bytes of padding: In fact, strongSwan has another classical flaw. The PKCS#1 v1.5 standard requires the number of padding bytes to be at least 8 [RFC2313, RFC2437]. Unfortunately, during our initial testing with *TH1*, we quickly realized that strongSwan does not check whether `PS` has a minimum length of 8, a flaw previously also found in other implementations [24]. Since `PS` is terminated with `0x00`, during symbolic execution, our initial *TH1* automatically generated test cases where some early byte of `PS` is given the value of `0x00`, and hence the subsequent symbolic bytes would be considered to be part of `AS`. And because strongSwan attempts to parse `AS` using an ASN.1 parser, this resulted in many paths enumerating different possible ASN.1 types with symbolic lengths. After finding this flaw, we have added additional constraints to *TH1* to guide the symbolic execution into not putting `0x00` in `PS`, which in the end resulted in a reasonable number of paths.

Lax ASN.1 length checks: Additionally, the weaknesses regarding algorithm parameter and algorithm OID also led to lenient handling of their corresponding length variables, u and y. This is the reason why *TH3* found several accepting paths, as the parser used during verification enumerated various combinations of values for u and y that it considers acceptable.

3) ***axTLS 2.1.3:*** axTLS is a very small footprint TLS library designed for resource-constrained platforms, which has been deployed in various system on chip (SoC) software stacks, *e.g.*, in Arduino for ESP8266[2], the Light Weight IP stack (LWIP)[3] and MicroPython[4] for various microcontrollers.

Unfortunately, the signature verification in axTLS is some of the laxest among all the recent implementations that we have tested. Its code is aimed primarily at traversing a pointer to the location of the hash value, without enforcing rigid sanity checks on the way. The various weaknesses in its implementation can lead to multiple possible exploits.

Accepting trailing bytes (CVE-2018-16150): We first found that the axTLS implementation accepts O that contains trailing bytes after the hash value, in order words, it does not enforce the requirement on the length of padding bytes, a classical flaw previously found in other implementations [5], [20], [27]. This is also why for both *TH1* and *TH2* there are multiple accepting paths.

Ignoring prefix bytes: On top of that, we found that this implementation also ignores the prefix bytes, including both BT and PB, which also contributes to the various incorrect accepting paths yielded by *TH1* and *TH2*. As shown in Snippet 2, this effectively means that the first 10 bytes of O can take arbitrarily any values. Such a logic deviates from what the standard prescribes [RFC2437], and as we will explain later in Section VI-8, an over-permissiveness like this can be exploited to forge signatures when e is small.

Snippet 2. Block Type and Padding skipped in axTLS 2.1.3

```
i = 10;/* start at the first possible non-padded byte */
while (block[i++] && i < sig_len);
size = sig_len - i;
/* get only the bit we want */
if (size > 0) {... ...}
```

Ignoring ASN.1 metadata (CVE-2018-16253): Moreover, we found that axTLS does not check the algorithm OID and parameter. In fact, through root-cause analysis, we found that this could be attributed to the parsing code shown in Snippet 3 below, which skips the entire `AlgorithmIdentifier` part of `AS` (achieved by `asn1_skip_obj()`), until it reaches the hash value (type `OCTET STRING`), making this even laxer than the flaws of not checking algorithm parameter previously found in other libraries [13], [27].

Snippet 3. Majority of ASN.1 metadata skipped in axTLS 2.1.3

```
if (asn1_next_obj(asn1_sig, &offset, ASN1_SEQUENCE) < 0
 || asn1_skip_obj(asn1_sig, &offset, ASN1_SEQUENCE))
  goto end_get_sig;

if (asn1_sig[offset++] != ASN1_OCTET_STRING)
    goto end_get_sig;
*len = get_asn1_length(asn1_sig, &offset);
ptr = &asn1_sig[offset];          /* all ok */

end_get_sig:
    return ptr;
```

Trusting declared lengths (CVE-2018-16149): Furthermore, using our approach, we have automatically found several test cases that could trigger memory errors at various locations of the axTLS source code. This is because given the various length variables in the ASN.1 structure that are potentially under adversarial control, the parser of axTLS, partly shown in Snippet 3, is too trusting in the sense that it uses the declared values directly without sanity checks, so one can put some absurd values in those lengths to try to trick the implementation into reading from illegal memory addresses and potentially crash the program. This is an example of CWE-130 (*Improper Handling of Length Parameter*).

This is also part of the reason why for *TH1*, it took more than 1 hour to finish the execution, as KLEE discovered many test cases that can trick the parsing code into reading z, the ASN.1 length of $H(m_s)$, from some symbolic trailing bytes, which led to several invocations of `malloc()` with huge sizes and hence the long execution time.

4) ***MatrixSSL 3.9.1:*** MatrixSSL requires $|n|$ to be a multiple of 512, so in our test harnesses, we have adjusted the size of the test buffer and padding accordingly. Interestingly, we have observed that MatrixSSL contains 2 somewhat different implementations of PKCS#1 v1.5 signature verification, one for verifying signatures on CRLs, and the other for certificates. Both are using a parsing-based verification approach. Why the

[2] https://github.com/esp8266/Arduino/tree/master/tools/sdk/lib

[3] https://github.com/attachix/lwirax

[4] https://github.com/micropython/micropython/tree/master/lib

two cases do not share the same signature verification function is not clear to us. Nevertheless, we have tested both of them, and to our surprise, one verification is laxer than the other, but both exhibit some forms of weaknesses.

Lax ASN.1 length checks: We first note that for both signature verification functions, their treatments of some of the length variables in AS are overly permissive. Quite the opposite of axTLS, we found that MatrixSSL does not fully trust the various ASN.1 lengths, and imposes sanity checks on the length variables. Those, however, are still not strict enough.

For the certificate signature verification, the first 2 ASN.1 lengths variables, w, and x (lengths of the two ASN.1 SEQUENCE in AS), are allowed ranges of values in the verification. For w, the only checks performed on it are whether it is in the long form, and whether it is longer than the remaining buffer containing the rest of O. Similarly, there exist some sanity checks on x but they are nowhere near an exact match warranted by a construction-based approach. The 2 accepting paths yielded by *TH3* are due to a decision being made on whether x matches exactly the length of the remaining SEQUENCE (OID and parameters) that had been consumed, which indicates whether there are extra bytes for algorithm parameters or not. However, this check is done with a macro psAssert(), which terminates only if HALT_ON_PS_ERROR is defined in the configuration, a flag that is considered to be a debugging option [23], not enabled by default and not recommended for production builds, meaning that many possible values of x, even if they failed the assertion, would still be accepted. When the length of the encoded OID is correct (*i.e.*, 5 for SHA-1), the length of algorithm parameters, y, is not checked at all.

For the CRL signature verification function, the treatments of length variables w, x, and y are also overly permissive, similar to what is done in certificate signature verification. On top of that, the checks on z the declared size of $H(m_s)$ in AS is also overly permissive, similar to those on w.

Comparing to a construction-based approach, these implementations are overly permissive and the weaknesses discussed allow some bits in O to take arbitrary any values, which means the verification is not as robust as it ideally should be.

Mishandling Algorithm OID: We found that for the CRL signature verification, there exists another subtle implementation weakness in how it handles the OID of hash algorithms.

As shown in the following snippet, upon finishing parsing the algorithm OID, the verification code would see whether the length of hash output given by the parsed algorithm matches what the caller of the verification function expects. However, since this is again done by the psAssert() macro, which as discussed before, does not end the execution with an error code even if the assertion condition fails, and the execution would just fall through. This explains the numerous accepting paths found by *TH2* and *TH3*.

Snippet 4. Checking Signature Hash Algorithm in MatrixSSL (CRL)

```
/** outlen := length of H(m) provided by caller,
   oi is the result of OID parsing **/
if (oi == OID_SHA256_ALG)
     {  psAssert(outlen == SHA256_HASH_SIZE);  }
else if (oi == OID_SHA1_ALG)
     {  psAssert(outlen == SHA1_HASH_SIZE);    }
... ...
else {  psAssert(outlen == SHA512_HASH_SIZE);  }
```

The implications of this flaw is that for the algorithm OID bytes (the length of which is subject to the checks discussed before), they can be arbitrarily any values, since in the end, it is the expected length of $H(m)$ provided by the caller of the verification function that dictates how the rest of the parsing would be performed. Hence the verification is overly permissive and one can get at most 9 arbitrary bytes in the OID part of O this way (*e.g.*, with $H()$ being SHA-256).

Besides, even if psAssert() would actually terminate with errors, the above implementation is still not ideal, as the assertion conditions are done based on the length of $H(m)$, not the expected algorithm. We note that the hash size and length of OID are not unique across hash algorithms. Since there are pairs of hash algorithms (*e.g.*, MD5 and MD2; SHA-256 and SHA3-256) such that (1) the length of their OIDs are equal, and (2) the length of their hash outputs are equal, the parser would consider algorithms in each pair to be equivalent, which can still lead to an overly permissive verification. Ideally, this should be done instead by matching the parsed OID against a caller provided expected OID.

5) ***GnuTLS 3.5.12:*** Based on our testing and root-cause analysis, GnuTLS is now using a construction-based approach in its PKCS#1 v1.5 signature verification code, which is a considerable improvement to some of its own vulnerable versions from earlier [25], [27]. This is also reflected in the small number of paths yielded by our test harnesses, even less than those that adopt a hybrid approach. Consequently, we choose this as the anchor for cross-validation.

6) ***Dropbear SSH 2017.75:*** Dropbear implements the SSH protocol, and uses *libtomcrypt* for most of the underlying cryptographic algorithms like the various SHA functions and AES. Interestingly, instead of relying on *libtomcrypt*'s RSA code, for reasons unbeknownst to us, Dropbear SSH has its own RSA implementation, written using the *libtommath* multiple-precision integer library. Based on our root-cause analysis, it appears that the PKCS#1 v1.5 signature verification implemented in the RSA implementation of Dropbear SSH follows the construction-based approach, hence it successfully cross-validated with the anchor and no particular weaknesses were found. In contrast to the bundled *libtomcrypt* which has some signature verification weaknesses (explained below), having its own RSA implemented actually helped Dropbear SSH to avoid some exploitable vulnerabilities.

Comparing to other implementations of construction-based verification (*e.g.*, BoringSSL), the *TH1* of Dropbear SSH took a significantly longer time to run, mainly due to the final comparison after constructing the expected I_v is done in the multiple-precision integer level, not with a typical memory comparison function like memcmp(). Nevertheless, it still managed to finish within a reasonable amount of time. As a side benefit, symbolic execution also covered part of the multiple-precision integer *libtommath* code.

7) ***libtomcrypt 1.16:*** Based on our test results, we found that libtomcrypt is also using a parsing-based approach, and its signature verification contains various weaknesses[5].

[5] Some of the weaknesses had been independently found by other researchers, leading to certain fixes being introduced in version 1.18.

Accepting trailing bytes: Similar to axTLS, libtomcrypt also has the classical flaw of accepting signatures with trailing bytes after $H(m_s)$, hence a practical signature forgery attack is possible when the public exponent is small enough. This is the reason why for *TH1* and *TH2*, there are 3 accepting paths.

Accepting less than 8 bytes of padding: Interestingly, libtomcrypt also has the classical flaw of not checking whether `PS` has a minimum length of 8, similar to strongSwan. Through root-cause analysis, we quickly identified the lax padding check as shown below. Give this verification flaw, to avoid scalability challenges due to symbolic padding bytes, we apply the same workaround to *TH1* as we did for strongSwan.

Snippet 5. Padding Check in libtomcrypt 1.16

```
for (i = 2; i < modulus_len - 1; i++)
    { if (msg[i] != 0xFF) { break; }  }
/* separator check */
if (msg[i] != 0) {
    /* There was no octet with hexadecimal value
        0x00 to separate ps from m. */
    result = CRYPT_INVALID_PACKET;
    goto bail;
}
/** ... start ASN.1 parsing at msg[i+1] ... **/
```

Lax AlgorithmIdentifier length check: Furthermore, despite the fact that *TH3* yielded only one accepting path, it turns out there is another subtle weakness in libtomcrypt. We found that in `AS`, the length x of `AlgorithmIdentifier` (the inner ASN.1 `SEQUENCE`) is checked only loosely, despite the constraints imposed on x by the verification code. This is because the constraints are mostly simple sanity and boundary checks such that x cannot be too small or too large, but the x is not required to match exactly to a concrete value (*i.e.*, 9 with explicit `NULL` parameter and $H()$ being SHA-1). This is partly because the ASN.1 parser used by libtomcrypt, *re-encodes* the bytes of an ASN.1 simple type that were just parsed, to calculate the actual length that was consumed. Hence, when given a child of ASN.1 `OID`, the length of the parent `SEQUENCE`, as in the case of `AlgorithmIdentifier`, was not checked strictly. This is also why for *TH2* it needed to change a handful of lines more, to workaround the re-encoding of OID which has decisions to be made for each byte, depending on whether it is less than 128 (short form) or not (long form).

Because the verification code would accept a range of values for x, this gives some bits in the middle of `AS` that one can choose arbitrary and is hence overly permissive.

8) ***mbedTLS 2.4.2:*** Based on the results of our testing, mbedTLS appears to be also using the parsing-based verification approach. The relatively larger number of paths from *TH1* and *TH3* can be attributed to the underlying ASN.1 parser, as there are various decisions (*e.g.*, whether the lengths are in the long form or not) to be made during parsing. We note that despite each of $\{$*TH1, TH2, TH3*$\}$ gave exactly one accepting path, only the paths extracted by *TH1* and *TH2* were successfully cross-validated with the other implementations. Upon close inspection of the one and only accepting path yielded by *TH3*, we realized it contains a subtle verification weakness, which was also caught by cross-validation.

Lax algorithm parameter length check: Interestingly, in mbedTLS 2.4.2, the checks imposed on y, the length of algorithm parameter, are in fact too lenient. Through root-cause analysis with CPT, we found that the only constraints imposed came from the parser, as shown in Snippet 6. There are 2 constraints, one is whether the most significant bit is on, which the parser uses to decide how it should obtain the actual length. The other one is whether the declared length is longer than what is remaining in the buffer.

Snippet 6. Only parsing and sanity checks imposed on y in mbedTLS 2.4.2

```
if( ( **p & 0x80 ) == 0 ) *len = *(*p)++;
else { ... ... }

if( *len > (size_t) ( end - *p ) )
    return( MBEDTLS_ERR_ASN1_OUT_OF_DATA );
```

Since after the parser consumed y, there would be 22 bytes left in the buffer (assuming no parameter bytes, $2 + 20$ for a SHA-1 hash), it turns out the verification code would accept any values of y not larger than 22, which allows some bits of `AS` to be arbitrarily chosen and is hence overly permissive.

9) ***BoringSSL 3112, BearSSL 0.4** and **wolfSSL 3.11.0:*** BoringSSL is a fork of OpenSSL, refactored and maintained by Google. We found its PKCS#1 v1.5 signature verification uses a hybrid approach. Everything before `AS` in O is handled and checked by a parser that scans through the buffer, and then `AS` is copied out. The verification code then constructs its own expected version of $\texttt{AS}_v$ using $H(m_v)$ and some hard-coded ASN.1 prefixes, and then compares $\texttt{AS}_v$ against `AS`. This observed behavior is consistent with what was reported earlier [13]. Consequently, the total number of paths are reasonably small, with each of $\{$*TH1, TH2, TH3*$\}$ yielding exactly one accepting path. BearSSL and wolfSSL both behaved quite similar to BoringSSL, and all 3 implementations successfully cross-validate against the anchor with no discrepancies observed. wolfSSL yielded more paths in *TH1* due to a slightly different handling of `PB`, and BearSSL yielded more paths in *TH2* due to extra handling of the case of absent parameter.

10) ***OpenSSL 1.0.2l** and **LibreSSL 2.5.4:*** We found that OpenSSL adopts a parsing-based verification approach, which partly explains why some higher number of paths were yielded by *TH2* and *TH3*. The slightly longer execution time of *TH3* can partly be attributed to the concretization workaround. Despite these, no verification weaknesses were found in this recent version of OpenSSL, which is perhaps unsurprising given that it had gone through years of scrutiny by security researchers [27]. LibreSSL is a fork of OpenSSL maintained by the OpenBSD community since 2014 after the infamous Heartbleed vulnerability. The two are actually quite similar when it comes to PKCS#1 v1.5 signature verification, both using a similar parsing-based approach and the test harnesses all yielded comparable numbers of execution paths.

11) ***PuTTY 0.7:*** We found that the PuTTY implementation of PKCS#1 v1.5 signature verification is highly reminiscent of a construction-based approach. The left-most 2 bytes of O containing `0x00` and `BT` are checked first, followed by a check on `PB` with an expected length (which depends on $|n|$), and then `AS` before $H(m_s)$ is checked against some hard-coded ASN.1 encoded bytes, and finally, $H(m_s)$ is checked. Cross-validation found no discrepancies and no signature verification weaknesses were detected.

Interestingly, even after sufficient rejection criteria has been hit (*e.g.*, `BT` is not `0x01`), the verification continues with other checks, until all has been finished and then an error would finally be returned. Since the later checks before the verification function returns do not alter a rejection return code back into an acceptance, this is not a verification weakness. We suspect this insistence on traversing the whole buffer containing O might be an attempt to avoid timing side channels.

However, as explained below with Example 2, such an implementation presents a small hurdle for symbolic execution, as the number of paths due to `if` statements (the series of checks) exhibits a multiplicative build-up, leading to a scalability challenge observed in our first round experiment with *TH1*. Consequently, we modified the source to adopt an '*early return*' logic, like a typical implementation of `memcmp()` would do. That is, once a sufficient rejection condition has been reached, the verification function returns with an error without continuing with further checks, so that the number of paths would build up additively. This explains why the number of lines changed in PuTTY is slightly higher than the others.

```
if (symBuf[0] != 0) ret = 0;
if (symBuf[1] != 1) ret = 0;
if (symBuf[2] != 2) ret = 0;
return ret;
```

```
if (symBuf[0] != 0) return 0;
if (symBuf[1] != 1) return 0;
if (symBuf[2] != 2) return 0;
```

Example 2: For number of execution paths, the snippet on right builds up additively, but the one on left does so multiplicatively.

*12) **OpenSSH 7.7**:* OpenSSH is another open source SSH software suite. For handling PKCS#1 v1.5 signatures, it relies on OpenSSL (calling `RSA_public_decrypt()`) to perform the RSA computation and process the paddings of O. Afterwards, it compares the `AS` returned by OpenSSL against its constructed version, hence it is somewhat of a hybrid approach. Cross-validation found no discrepancies and no weaknesses were detected in the verification.

Interestingly, instead of simply using `memcmp()`, the comparison against the constructed `AS` is done using a custom constant time comparison, as shown below:

```
/** p1,2 point to buffers of equal size(=n) **/
for (; n > 0; n--) ret |= *p1++ ^ *p2++;
return (ret != 0);
```

This explains why *TH3* found in total only 2 paths of relatively larger constraints, as such a timing safe comparison would aggregate (with `OR`) the comparison (with `XOR`) of each byte in the two buffers. Semantically, the 2 execution paths mean either all length variables u, w, x, y, z in *TH3* match their expected values exactly, or at least one of them does not.

VI. Exploiting Our New Findings

Here we discuss how to exploit the several weaknesses presented in the previous section. For ease of discussion, we focus on SHA-1 hashes, but the attacks can be adapted to handle other hash algorithms by adjusting the lengths of appropriate components. Though low-exponent RSA public keys are rarely seen in the Web PKI nowadays [18], there are specific settings where low-exponent keys are desired (*e.g.*, with extremely resource-constrained devices). Historically, a small public exponent of $e = 3$ has been recommended for better performance [RFC3110], and there are key generation programs that still mandate small public exponents [7].

*1) **Signature forgery against Openswan**:* The flaw of *ignoring padding bytes* effectively means Openswan would accept a malformed O' in the form of

`0x00 || 0x01 || GARBAGE || 0x00 || AS` ,

which can be abused in a manner similar to the signature forgery attack exploiting the weakness of not checking algorithm parameters found in some other implementations as discussed in previous work [27].

This has serious security implications. We note that in the context of IPSec, the key generation program `ipsec_rsasigkey` forces $e = 3$ without options for choosing larger public exponents [7]. Since the vulnerable signature verification routine is used by Openswan to handle the `AUTH` payload, the ability to forge signatures might enable man-in-the-middle adversaries to spoof an identity and threaten the authentication guarantees delivered by the `IKE_AUTH` exchange when RSA signature is used for authentication.

Given the implementation flaw allows for certain bytes *in the middle* of O' to take arbitrarily any values, the goal of the attack is to forge a signature $S' = (k_1 + k_2)$, such that when the verifier computes $O' = {S'}^3 = (k_1+k_2)^3 = {k_1}^3+3{k_1}^2k_2+3{k_2}^2k_1+{k_2}^3$, the following properties would hold:

1) the *most significant bits* of ${k_1}^3$ would be those that need to be matched exactly *before* the unchecked padding bytes, which is simply `(0x00 || 0x01)`;
2) the *least significant bits* of ${k_2}^3$ would become those that need to be matched exactly *after* the unchecked padding bytes, which is simply `(0x00 || AS)`;
3) the *most significant bits* of ${k_2}^3$ and the *least significant bits* of ${k_1}^3$, along with $3{k_1}^2k_2 + 3{k_2}^2k_1$, would stay in the unchecked padding bytes.

One influential factor to the success of such attack is whether there are enough unchecked bytes for an attacker to use. An insufficient amount would have the terms of expanding $(k_1 + k_2)^3$ overlapping with each other, make it difficult for the three properties to hold. However, since the flaw we are exploiting is on the handling of padding bytes, the number of which grows linearly with $|n|$, assuming the same public exponent, a longer modulus would actually contribute to the attacker's advantage and make it easier to forge a signature. Specifically, assuming SHA-1 hashes and $e = 3$, given $|n| \geq 1024$ bits, it should be easy to find k_1 and k_2 that satisfy the three properties without worrying about overlaps.

***Finding** k_1.* The main intuition used is that a feasible k_1 can be found by taking a cubic root over the desired portion of O'. For instance, in the case of $|n| = 1024$ bits, `0x00 || 0x01 || 0x00 ... 0x00` is simply 2^{1008} (with 15 zero bits in front), hence a simple cubic root would yield a $k_1 = 2^{336}$.

In the more general cases where $|n| - 15 - 1$ is not a multiple of 3, the trailing garbage could be used to hide an over-approximation. One can first compute $t_1 = \lceil \sqrt[3]{2^{|n|-15-1}} \rceil$ and then sequentially search for the largest possible r such that $((t_1/2^r + 1) \cdot 2^r)^3$ gives `0x00 || 0x01 || GARBAGE`. Then k_1 would be $(t_1/2^r + 1) \cdot 2^r$. This is to make as many ending bits of k_1 to be zero as possible, to avoid overlapping terms in the expansion of $(k_1 + k_2)^3$. For example, when $|n| = 2048$ bits, we found $r = 676$ bits and $k_1 = 3 \cdot 2^{676}$.

***Finding** k_2.* The intuition is that to get `(0x00 || AS)` with ${k_2}^3$, the modular exponentiation can be seen as computed over a much smaller n'' instead of the full modulus n. While finding $\phi(n)$ reduces to factorizing n, which is believed to be impractical when n is large, finding $\phi(n'')$ can be quite easy.

One can consider $S'' =$ (0x00 || AS) and $n'' = 2^{|S''|}$, where $|S''|$ is the size of AS in number of bits plus 8 bits for the end of padding 0x00.

Now k_2 has to satisfy ${k_2}^e \equiv S'' \pmod{n''}$. Since n'' is a power of 2, we can guarantee k_2 and n' are coprime by choosing an odd numbered S'' with a fitting hash value. Also, $\phi(n'') = \phi(2^{|S''|}) = 2^{|S''|-1}$.

One can then use the Extended Euclidean Algorithm to find f such that $ef \equiv 1 \pmod{2^{|S''|-1}}$. With f found, k_2 would simply be ${S''}^f \pmod{n''}$.

We have implemented attack scripts assuming $e = 3$ and SHA-1 hashes, and were able to forge signatures that would be successfully verified by Openswan 2.6.50 given any $|n| = 1024$ and $|n| = 2048$ moduli.

2) ***Signature forgery (1) against strongSwan****:* The flaw of *not checking algorithm parameter* can be directly exploited for signature forgery following the algorithm given in [27] (which is very similar to the attack we described previously against Openswan). Assuming $e = 3$, $|n| = 1024$ bits and SHA-1 hashes, the expected iterations required to brute-force a fake signature is reported to be 2^{21} [27].

3) ***Signature forgery (2) against strongSwan****:* Likewise, the flaw of *accepting trailing bytes after OID* can be exploited following the steps used in the forgery attack against Openswan as described before, by adjusting what ${k_1}^3$ and ${k_2}^3$ represent. Under the same parameter settings, it should require a comparable number of iterations as *signature forgery (1)* does discussed above.

4) ***Signature forgery (3) against strongSwan****:* Interestingly, the flaw of *accepting less than 8 bytes of padding* can be exploited together with the algorithm parameter flaw to make it easier to forge signatures. In fact, the two flaws together means such an O' with no paddings at all would be accepted:

```
/** all numbers below are hexadecimals **/
00 01 00 30 7B 30 63 06   05 2B 0E 03 02 1A 05 5A
GARBAGE  04 16 SHA-1(m')
```

The length of algorithm parameter 0x5A is calculated based on $|n|$ (in this case 1024 bits) and the size of hash. Then by simply adjusting what ${k_1}^3$ and ${k_2}^3$ represent in the attack against Openswan, given $e = 3$ and $|n| \geq 1024$ bits, the forgery will easily succeed. We implemented this new variant of attack and confirmed that the fake signatures generated actually work.

5) ***Signature forgery (4) against strongSwan****:* Similarly, the forgery attack exploiting trailing bytes after OID could also benefit from the absence of padding, as an O' like the followings would be accepted by strongSwan:

```
/** all numbers below are hexadecimals **/
00 01 00 30 7B 30 63 06   5F 2B 0E 03 02 1A
GARBAGE 05 00 04 16 SHA-1(m')
```

The length of algorithm OID 0x5F is calculated based on $|n|$ (in this case 1024 bits) and the size of hash. The attack against Openswan would work here as well, simply by adjusting what ${k_1}^3$ and ${k_2}^3$ represent. Signature forgery would again easily succeed given $e = 3$ and $|n| \geq 1024$ bits. We have also implemented this new attack variant and confirmed that the fake signatures generated indeed work.

6) ***Signature forgery (1) against axTLS****:* Given that there exist performance incentives in using small exponents with the kinds of resource-constrained platforms that axTLS targets, a practical signature forgery attack as described in [27] could be made possible by the flaw of *accepting trailing bytes*. Specifically, when $|n| = 1024$, assuming $e = 3$ and SHA-1 hashes, the expected number of trials before a successful forgery is reported to be around 2^{17} iterations, which takes only several minutes on a commodity laptop [27]. As a larger $|n|$ would allow for more trailing bytes, hash algorithms that yield longer hashes could be attacked similarly, *e.g.*, assuming $e = 3$ and SHA-256 hashes, a modulus with $|n| = 2048$ bit should easily yield a successful forgery. Similarly, such an attack would also work against a larger public exponent with an accordingly longer modulus.

7) ***Signature forgery (2) against axTLS****:* Separately, the weakness of *ignoring ASN.1 metadata* as shown in Snippet 3, can also be exploited for a low-exponent signature forgery. Due to the majority of AS being skipped over, axTLS would accept an O' like this:

```
/** all numbers below are hexadecimals **/
00 01 FF FF FF FF FF FF   FF FF 00 30 5D 30 5B
GARBAGE 04 16 SHA-1(m')
```

where the lengths 0x5D and 0x5B are calculated based on $|n|$ and size of hash to make sure the skipping would happen correctly. Then the forgery attack against Openswan described before can be easily adapted to work here by adjusting what ${k_1}^3$ and ${k_2}^3$ represent. Given $|n| \geq 1024$, forgery should easily succeed. We have tested the adapted attack script and the forged signatures it generates indeed worked on axTLS.

8) ***Signature forgery (3) against axTLS****:* Knowing that axTLS also *ignores prefix bytes* as shown in Snippet 2, the *signature forgery (1)* described above which exploits *unchecked trailing bytes* can be made even easier to succeed, by making the first 11 bytes all 0 (including the end of padding indicator). Adapting the analysis from previous work [27], the signature value O is essentially a number less than 2^{935} (assuming $|n| = 1024$, the first 88 bits are all zeros, with 2 additional zero bits from the first 0x30 byte of AS). The distance between two consecutive perfect cubes in this range is

$$\begin{aligned} k^3 - (k-1)^3 = 3k^2 - 3k + 1 &< 3 \cdot 2^{624} - 3 \cdot 2^{312} + 1 \\ &< 2^{626} \qquad\qquad (\because k^3 < 2^{935}) \qquad (1) \end{aligned}$$

which is less than the 656 bits that an attacker can choose arbitrarily (46 bytes are fixed, due to the 35-byte AS containing a desired SHA-1 hash and the 11 bytes in front), so a signature forgery should easily succeed, by preparing an attack input O' containing hash of an attacker-chosen m', and the attack signature S' can be found by simply taking the cubic root of O'. Once the verifier running axTLS 2.1.3 received S', it would compute $O' := {S'}^3 \mod n$, and despite O' being malformed, the verification would go through.

9) ***Signature forgery (4) against axTLS****:* Furthermore, the weakness of *ignoring ASN.1 metadata*, can be exploited together with the previous attack, to make the signature forgery even easier. The intuition is that, knowing the parsing code would skip over the ASN.1 prefix (the two 0x30 ASN.1 SEQUENCE) according to the length declared, an attacker can spend the minimal number of bytes on AS to keep the parser entertained, with an O' like this:

```
/** all numbers below are hexadecimals **/
00 00 00 00 00 00 00 00   00 00 00 30 00 30 00 04
H().size   H(m')    TRAILING
```

and spend the gained extra free bytes at the end as trailing ones. While for SHA-256 and $|n| = 1024$, a signature forgery attack exploiting *only* trailing bytes has the expected iterations of about 2^{145} [27], however, if we use this joint attack strategy instead, this bound can be pushed down much lower and the attack becomes practical. Specifically, assuming SHA-256, the joint attack strategy would have $11 + 6 + 32 = 49$ bytes fixed, and 79 trailing bytes (632 bits) at the end that the attacker can choose arbitrarily, more than the bound of 626 bits on the distance between two perfect cubes from eq. (1), so a forgery should easily succeed by taking the cubic root as described before. We have implemented attack scripts and successfully performed this new variant of signature forgery on axTLS 2.1.3 with $e = 3, |n| = 1024$ and for both SHA-1 and SHA-256.

10) ***Denial of Service against axTLS****:* We further note that because of the *trusting* nature of the parser in axTLS, an illegal memory access attack against axTLS with absurd length values is also possible, which might crash the verifier and result in a loss of service. Specifically, following the previous forgery attack, we prepared an attack script that generates signatures which would yield a z (the length of hash) of `0x84`, and the illegal memory access induced by this absurd value had successfully crashed the verifier in our experiments.

We further note that such a denial of service attack can be even easier to mount than a signature forgery in the context of certificate chain verification. This is due to the fact that axTLS verifies certificate chains in a bottom-up manner, which contributes to an attacker's advantage: even if low-exponent public keys are rarely used by real CAs in the wild, to crash a verifier running axTLS, one can purposefully introduce a counterfeit intermediate CA certificate that uses a low-exponent as the j-th one in the chain, and forge a signature containing absurd length values as described above and put it on the $(j + 1)$-th certificate. Due to the bottom-up verification, before the code traverses up the chain and attempts to verify the j-th counterfeit certificate against the $(j - 1)$-th one, it would have already processed the malicious signature on the $(j + 1)$-th certificate and performed some illegal memory access. While a bottom-up certificate chain verification is not inherently wrong, but because of the weaknesses in the signature verification, the bottom-up design has an unexpected exploitable side effect. **This highlights why a signature verification code needs to be robust regardless of the choice of e.**

11) ***Signature forgery (1) against libtomcrypt****:* Just like the flaw of *accepting trailing bytes* in axTLS, the same flaw in libtomcrypt 1.16 can also be exploited in a signature forgery attack if the e is small enough and $|n|$ is large enough, following the same attack algorithm described in [27].

12) ***Signature forgery (2) against libtomcrypt****:* We note that the flaw of *accepting less than* 8 *bytes of padding* found in libtomcrypt 1.16 also has serious security implications. Combining this with the attack exploiting trailing bytes, the low-exponent signature forgery can be made even easier. Specifically, an attacker can craft an O' like this:

```
/** all numbers below are hexadecimals **/
00 01 00 || AS || TRAILING || EXTRA TRAILING
```

The intuition behind is that one can shorten the padding as much as possible, and spend the extra bytes at the end. Assuming $|n| = 1024, e = 3$ and $H()$ is SHA-1, this attack has 38 bytes fixed, and hence $1024 - 38 \cdot 8 = 720$ bits that the attacker can choose arbitrarily. Since in this case, O' is essentially a number $< 2^{1010}$, the distance between two consecutive perfect cubes in this range is

$$\begin{aligned} k^3 - (k-1)^3 = 3k^2 - 3k + 1 &< 3 \cdot 2^{674} - 3 \cdot 2^{337} + 1 \\ &< 2^{676} \qquad\qquad (\because k^3 < 2^{1010}), \end{aligned}$$

which is less than the 720 bits that can be chosen arbitrarily, so a signature forgery would succeed easily. We have implemented an attack script and verified the success of such a signature forgery attack against libtomcrypt 1.16.

13) ***Other weaknesses****:* We note that not all the weaknesses found can immediately lead to a practical Bleichenbacher-style low-exponent signature forgery attack. For example, even though the other weaknesses in mbedTLS 2.4.2, MatrixSSL 3.9.1 and libtomcrypt 1.16 regarding *lax length variable checks* allow for some bits to take arbitrary any values, given that the number of free bits gained due to those weaknesses appear to be somewhat limited, it is not immediately clear how to exploit them for signature forgery. Nevertheless, those implementations are accepting signatures that should otherwise be rejected, which is less than ideal and might potentially be taken advantage of when combined with some other unexpected vulnerabilities in a specific context.

VII. Disclosure and Fixes

In an effort of responsible disclosure, we have notified vendors of the weak implementations so that they can have their signature verifications hardened. CVEs are requested and assigned selectively on the basis that a weakness can lead to immediate practical attacks as outlined above. Developers of *MatrixSSL* have acknowledged and confirmed our findings, and are currently implementing fixes. *strongSwan* has fixed the problems since version 5.7.0 and released further patches for older versions. *Openswan* has fixed the exploitable weakness since their 2.6.50.1 release and incorporated one of our forged signatures into their unit tests. *libtomcrypt* developers have created a ticket regarding the parser weakness and are currently investigating it. We developed a patch for *axTLS* and tested it with our approach before releasing it, and our patch has been incorporated by the ESP8266 port of axTLS. At the time of writing, we are awaiting responses from the vendor of *mbedTLS* and upstream maintainer of axTLS.

VIII. Related Work

Attacking PKCS#1 v1.5 implementations. Variants of implementation flaws in signature verification which enable possible forgery attacks have been found in a variety of software over the years [3], [5], [13], [20], [24], [25], [27], though the code-level analysis and discovery process were mostly based on manual inspection. We learn from these previous discoveries and demonstrate how a principled approach can be used to find subtle verification weaknesses and new variants of flaws.

Another class of prominent attacks on PKCS#1 v1.5 implementations is the padding oracle attacks (POAs) [8], [10], [11], [22]. The two classes of attacks capitalize on different issues. Signature forgery is made possible primarily due to overly permissive input parsing logic that deviates from what the specification mandates, an example of semantic correctness issues. POAs typically exploit the leakage provided by some error messages observable by the adversary. Padding oracles have also been found on deployments of symmetric key block ciphers [6], [8]. Specifications are often implicit and sometimes underspecified on how to prevent padding oracles.

Testing semantic correctness. As tools like libFuzzer and AFL as well as infrastructures like OSS-Fuzz are becoming prolific at finding memory access and runtime errors, the research community has seen an increased interests on identifying semantic level defects. In the absence of a test oracle that generates a correct output given an input [9], many research efforts resort to the principle of differential testing [19].

Fuzzing has been used to analyze the semantic correctness of TLS implementations [9]. Fuzzing can also be combined with the L* algorithm to extract finite state machines (FSM) out of TLS implementations, and semantic flaws can be found by analyzing the FSM [17]. There is also a framework that enable flexible configuration of message variables and sequences for evaluating TLS libraries and servers [29]. Fuzzing with differential testing has also been used to investigate the semantic correctness of implementations of X.509 certificate chain validation [12], [16]. The most relevant effort to this paper is SymCerts [15], where symbolic execution is used with differential testing to analyze implementations of X.509 certificate chain validation. However, their test case concretization strategy is not directly applicable to PKCS#1 v1.5.

IX. Conclusion

In this paper, we propose to enhance symbolic execution with meta-level search and constraint provenance tracking, for automatically generating concolic test cases and easier root-cause analysis. As a demonstration, we analyzed 15 open-source implementations of PKCS#1 v1.5 signature verification and found semantic flaws in 6 of them. We plan to publicly release the relevant source code and artifacts like extracted path constraints, so other researchers and practitioners can reproduce our work and leverage it to test other implementations.

Acknowledgments. We thank the reviewers, especially our shepherd Deian Stefan, for their insightful comments and suggestions. Special thanks to Durga Keerthi Mandarapu of IIT Hyderabad for analyzing some of the attack algorithms with us. This work was supported in part by NSF grant CNS-1657124, United States ARO grant W911NF-16-1-0127, as well as a grant from the NSA-sponsored Science of Security Lablet at North Carolina State University.

References

[1] "OSDI'08 Coreutils Experiments," http://klee.github.io/docs/coreutils-experiments.

[2] "Tutorial on How to Use KLEE to Test GNU Coreutils," https://klee.github.io/tutorials/testing-coreutils.

[3] *CVE-2006-4790*, 2006 (accessed Aug 01, 2018), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4790.

[4] *OpenSSL Security Advisory [5th September 2006]*, 2006 (accessed Aug 02, 2018), https://www.openssl.org/news/secadv/20060905.txt.

[5] *CVE-2006-4340*, 2006 (accessed Jul 18, 2018), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4340.

[6] *CVE-2016-0736*, 2016 (accessed Nov 01, 2018), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0736.

[7] *Ubuntu Manpage: ipsec_rsasigkey*, (accessed Aug 21, 2018), http://manpages.ubuntu.com/manpages/bionic/man8/ipsec_rsasigkey.8.html.

[8] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient padding oracle attacks on cryptographic hardware," in *Advances in Cryptology – CRYPTO 2012*, 2012, pp. 608–625.

[9] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE Symposium on Security and Privacy*, 2015.

[10] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1," in *Annual International Cryptology Conference*. Springer, 1998, pp. 1–12.

[11] H. Böck, J. Somorovsky, and C. Young, "Return of bleichenbacher's oracle threat (ROBOT)," in *27th USENIX Security Symposium*, 2018.

[12] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *IEEE S&P*, 2014, pp. 114–129.

[13] Bugzilla, *RSA PKCS#1 signature verification forgery is possible due to too-permissive SignatureAlgorithm parameter parsing*, 2014 (accessed Jul 18, 2018), https://bugzilla.mozilla.org/show_bug.cgi?id=1064636.

[14] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008, pp. 209–224.

[15] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "SymCerts: Practical Symbolic Execution For Exposing Noncompliance in X. 509 Certificate Validation Implementations," in *IEEE S&P*, 2017, pp. 503–520.

[16] Y. Chen and Z. Su, "Guided differential testing of certificate validation in ssl/tls implementations," in *ESEC/FSE 2015*, pp. 793–804.

[17] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *USENIX Security Symposium*, 2015, pp. 193–206.

[18] A. Delignat-Lavaud, M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie, "Web PKI: Closing the Gap between Guidelines and Practices." in *NDSS*, 2014.

[19] R. B. Evans and A. Savoia, "Differential testing: A new approach to change detection," in *ESEC-FSE companion '07*, 2007, pp. 549–552.

[20] H. Finney, *Bleichenbacher's RSA signature forgery based on implementation error*, 2006 (accessed Jul 06, 2018), https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html.

[21] J. Galea, S. Heelan, D. Neville, and D. Kroening, "Evaluating manual intervention to address the challenges of bug finding with KLEE," 2018. [Online]. Available: http://arxiv.org/abs/1805.03450

[22] S. Gao, H. Chen, and L. Fan, "Padding Oracle Attack on PKCS#1 V1.5: Can Non-standard Implementation Act As a Shelter?" in *Proceedings of the 12th International Conference on Cryptology and Network Security - Volume 8257*, 2013, pp. 39–56.

[23] INSIDE Secure, *MatrixSSL Developers Guide*, 2017 (accessed Jul 21, 2018), https://github.com/matrixssl/matrixssl/blob/master/doc/matrixssl_dev_guide.md#debug-configuration.

[24] Intel Security: Advanced Threat Research, *BERserk Vulnerability – Part 2: Certificate Forgery in Mozilla NSS*, 2014 (accessed Jul 06, 2018), https://bugzilla.mozilla.org/attachment.cgi?id=8499825.

[25] S. Josefsson, *[gnutls-dev] Original analysis of signature forgery problem*, 2006 (accessed Jul 21, 2018), https://lists.gnupg.org/pipermail/gnutls-dev/2006-September/001240.html.

[26] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[27] U. Kühn, A. Pyshkin, E. Tews, and R. Weinmann, "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.*, 2008.

[28] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *ACM PLDI '12*, pp. 193–204.

[29] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM CCS*, 2016, pp. 1492–1504.