2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 979-8-3503-8199-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/TrustCom60117.2023.00117

The Broken Verifying: Inspections at Verification Tools for Windows Code-Signing Signatures

Guangqi Liu*†, Qiongxiao Wang‡, Cunqing Ma*, Jingqiang Lin§, Yanduo Fu*, Bingyu Li¶ Dingfeng Ye*
*State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
Email:{liuguangqi,macunqing,fuyanduo,yedingfeng}@iie.ac.cn
†School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
‡Beijing Certificate Authority Co., Ltd., Beijing, China
Email:wangqiongxiao@bjca.org.cn
§School of Cyber Security, University of Science and Technology of China, Hefei, China
Email:linjq@ustc.edu.cn
¶School of Cyber Science and Technology, Beihang University, Beijing, China
Email:libingyu@buaa.edu.cn

***Abstract*—Terminal users can deploy verification tools to verify Windows code-signing signatures and check their details (signing time, certificate chain, etc). Some representative verification tools are also adopted in related studies, which take tools' outputs as contributing factors to analyse malicious software or certificate ecosystems. However, as code-signing signature verification is related to multiple dimensions, such as certificate status and system policies, getting accurate signature status and details is essential but rather complicated. And performance of different tools in verifications has not been well studied and compared with.**

We provide a novel methodology to inspect Windows code-signing verification tools, checking that if they print consistent results and details. We choose four representative tools to verify massive samples (more than 26 million) and collect their outputs. During the verification, we deploy a two-step verification method, which efficiently excludes 78.8% of samples (not signed). We write scripts to read each line of outputs, learning tools' output structures. Then we can precisely locate and extract interested code-signing fields from outputs. After that, we compare these essential fields from different tools, and analyze inconsistent cases. Finally, we present three types of inconsistent cases: verifying neglect, timestamp disturbance, and compatibility/robustness issues. We find some verification tools may assert code-signing signatures as invalid due to external factors, such as unexpected signing or invalid timestamp.

***Index Terms*—Windows code signing, signature verification, tool inspection**

I. Introduction

In Windows platform, code-signing signatures [19] can protect software from being tampered with and provide non-repudiation properties during software distributions. Software with valid signatures is easier to be trusted by the system or anti-virus programs [8] [7]. Therefore, developers are willing to append code-signing signatures into their products to improve credibility. On the other hand, malware developers expect to make up valid signatures to bypass system checking [7] [9] [10]. Verifying code-signing signatures, and getting accurate signature status or details seem straightforward, they are rather complicated. Firstly, signatures can be embedded into a portable executable (PE) [25] file (.exe, .dll, etc.) named *Authenticode* [20] or exist in a signed *Catalog* file (.cat) [21]. Secondly, the signature status (valid, invalid, not signed, etc.) is related to multiple dimensions, such as system clocks, system policies, and the status of signing certificates. And to construct complete certificate chains, verification programs may need to download some absent certificates from Internet. Thirdly, during the creation of signatures, a timestamp from trusted providers could be attached to the signature segment to extend the signature's lifetime, which also complicates signature verifications.

There are some popular verification tools available to verify code-signing signatures. They usually invoke Windows API (e.g. *func WinVerifyTrust* from *wintrust.h*) to verify code-signing signatures. For example, *Sigcheck* [12] is a third-party tool recommended by Microsoft. It is also integrated in the online malware-analysis platform VirusTotal[1]. *Signtool* [22] is an official tool integrated in Windows SDK. These tools serve as a supplement outside original Windows defence mechanisms, providing rich details about code-signing signatures (signature status, certificate chain parsing, timestamp, etc). Hence their outputs are widely adopted in related studies, to analyze code-signing issues (e.g. certifying malware or potentially unwawnted programs (PUP) [9] [7], vulnerabilities in verifications [8], and code-signing economics/ecosystems [10] [11]). However, in these studies, verification tools are directly used, without deep inspections towards tools themselves. If verification tools make mistakes, their users or integrated platform (e.g. VirusTotal) will get incorrect results and details, then affect further analysis for samples. Furthermore, except for tools that invoke Windows API, some open-source tools utilizes open-source libraries to verify code-signing signatures, such as *Osslsigncode* [13]. It is also valuable to check that if open source tools performance as well as Windows API when they deal with code-signing signatures.

In this paper, we provide a novel methodology to in-

Qiongxiao Wang is the corresponding author.

[1]https://www.virustotal.com

2324-9013/23/$31.00 ©2023 IEEE
DOI 10.1109/TrustCom60117.2023.00117

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

spect Windows code-signing verification tools, checking that if they print consistent results and details in verification. We raise a universal pipeline, and select four representative verification tools (*Sigcheck*, *Signtool*, *AnalyzePESig* [3], and *Osslsigncode*) to be inspected. In addition to signature status, these tools also print details about signing certificates and timestamps. We divide the methodology into three stages. In stage one, we download more than 26 million samples (13.42 TB) from VirusShare[2] as our sample source. Then, we verify all the samples with selected tools. To improve efficiency, we divide the verification procedure into two steps, which excludes 78.8% of samples (not signed) in the first step. We finally get 70.7 GB verification outputs, belonging to 5.5 million filtered samples. In stage two, we analyze these outputs and figure out their structures. As some verification tools are closed-source programs, we cannot automatically read and understand their outputs without manual work. So we write python scripts to recognize each line in outputs and collect 383 kinds of output sequences from all outputs. Then we acquire structure knowledge about these outputs. In stage three, with acquired output sequences, we locate and extract interested code-signing fields from outputs, and then compare them with each other, checking that if all tools print consistent results and details. Finally, we present three types of inconsistent cases: 1) Windows Catalog signatures will be neglected sometimes if a tool verifies Authenticode signatures first. 2) Windows API processes a more strict strategy for timestamp, while *Osslsigncode* does not. 3) Compatibility and robustness issues will lead to inconsistent outputs. For case 1 and 2, samples all contain valid code-signing signatures, but they are asserted to be invalid (by mistake or by design). We report our observation to related developers. In some way, our analysis demonstrates the complexity of Windows code-signing signature verification. Our methodology raises a universal pipeline to understand and compare different verification tools. It is also suitable for other code-signing signature verification tools such as Powershell and Signify.

In summary, we make the following contributions:

- We raise a universal pipeline to understand and compare different Windows code-signing verification tools, checking that if they print consistent results and details in verification.
- We measure four representative tools with more than 26 million samples. We design a two-step verification method. It efficiently excludes 78.8% of samples (not signed). Meanwhile, we acquire a specialized dataset for code-signing studies and publish its metadata in our Github webpage[3].
- We recognize each text line in outputs, and summarize output structures. Structure knowledge allows us to precisely extract interested fields from tools' outputs.
- We compare interested fields from different tools' outputs and analyze three types of inconsistent cases, then report them to related developers.

[2]https://www.virusshare.com

[3]https://github.com/samKid3000/code-signing-sample-metadata

The rest of the paper is organized as follows. We introduce preliminaries in Section II. The methodology and corresponding data are presented in Section III and Section IV, respectively. We show analyses and discussions in Section V. Then we display related works in Section VI. Section VIII concludes this work.

II. Preliminaries

We introduce pertinent preliminaries about Windows code-signing and selected verification tools.

A. Windows Code-Signing

Authenticode Signature. Authenticode [15] is based on Public-Key Cryptography Standards (PKCS) #7 signed data and X.509 certificates to bind an Authenticode-signed binary to the identity of a software publisher. We illustrate the PE file format and Authenticode signature format in Figure 1. An Authenticode signature is embedded in a Windows PE file (.exe, .dll, etc.), in a location specified by the Certificate Table entry in Optional Header Data Directories. In the figure, an Authenticode hash value is calculated by PE objects with white background, then the software publisher signs the hash value with its code-signing certificates. The signature structure should contain the whole certificate chain of signers. It may contain a timestamp signed by a trusted timestamping authority (TSA). And PKCS #7 signed data structures are designed to support multiple signatures. For example, it may contain a signature signed with the SHA1 algorithm to satisfy outdated Windows systems and another signature signed with the SHA256 algorithm to satisfy current systems.

Digitally-Signed Catalog File. A digitally-signed catalog file [21] (.cat) can provide a digital signature for an arbitrary collection of files. Digest values for executable files are listed and signed in the catalog file. All executable files that participate in the catalog share a single signature. Then there is no need to sign files one by one like Authenticode.

Windows systems install catalog files to the CatRoot directory under the system directory, e.g., *%SystemRoot%\System32\CatRoot*. Catalog files are usually used to set signatures for driver packages. A catalog file will be automatically installed to the *CatRoot* when the driver package is staged to the *Driver Store* directory and will be automatically uninstalled from the *CatRoot* when the driver package is removed from the *Driver Store* directory. It should not be added to or removed from that directory manually.

Timestamp. If a TLS certificate expires, the certificate will be rejected by TLS clients. However if end-users reject software signed by expired code-signing certificates, all signatures only keep valid when the certificate is valid. This is unrealistic for software distributions. Timestamp [23] is introduced to solve this problem. Timestamp can prove that a signature is signed during its signing certificate's validity period. Then the signature should keep valid even when the signing certificate expires. To achieve this goal, the timestamp should be signed by a TSA.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

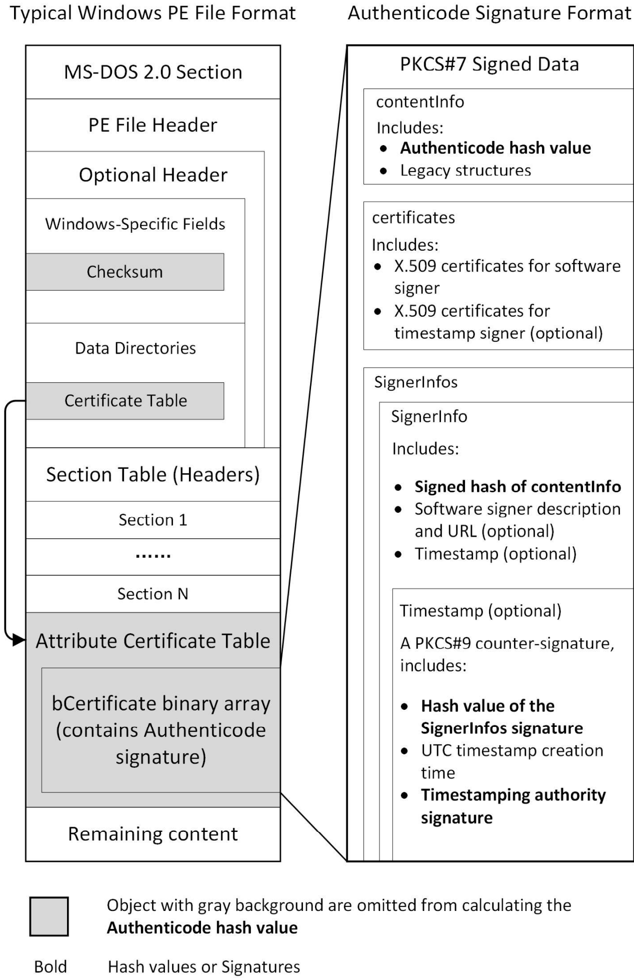

Fig. 1: PE File Format and Authenticode Signature Format

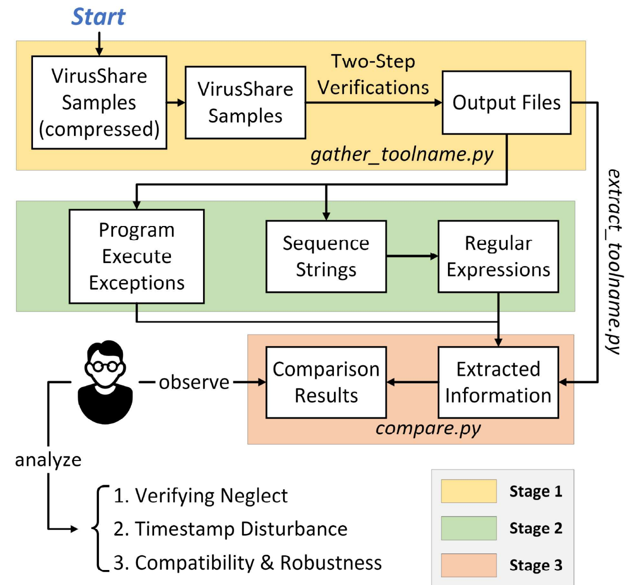

Fig. 2: Full Pipeline of Our Methodology

B. Verification Tools

We show the overview of verification tools in Table I. Within four verification tools, *Sigcheck (v2.82)* [12], *Signtool (v10.0.18362.1)* [22], and *AnalyzePESig (v0.0.0.5)* [3] invoke Windows API (e.g. *func WinVerifyTrust* from *wintrust.h*) to verify Authenticode and Catalog signatures. *Osslsigncode (v2.2)* [13] verifies signatures by open-source libraries such as OpenSSL and cURL. *Osslsigncode* cannot automatically verify Catalog signatures. So we just use it to verify Authenticode signatures in this paper. All four tools print rich details when verifying signatures, such as signing certificates (common name, serial number, fingerprint, validity period, etc), timestamp and timestamp certificates, reasons for invalid signatures.

III. Methodology and Implementation

We illustrate the full pipeline of our methodology in Figure 2. Data naming is described in Table II. We present our methodology with the following sketch:

Stage 1: We download samples (e.g. *Virusshare_md5val*) from VirusShare as original inputs, and design a two-step verification procedure to deal with these samples. Then for each filtered sample, we acquire its four verification output files (e.g. *Virusshare_md5val_Sigcheck.txt*).

Stage 2: We read each line in output files to learn tools' output structures. Then for each tool, we gather its sequence string list (e.g. *Sequence_Sigcheck.txt*) and then summarize its outputs' regular expression. We also record specific program-execute exceptions.

Stage 3: We locate and extract specific code-signing fragments from output files through outputs' structure knowledge. Finally, for each filtered sample, we compare its four extracted information (e.g. *Virusshare_md5val_extract_Sigcheck.json*).

We mainly present the methodology, implementation, and challenges in this section. Corresponding data collected or generated in the methodoloty is shown in next section.

A. Stage 1: Samples and the Two-Step Verifications

1) Sample Source: Until now, there is no specialized public dataset providing PE files with code-signing signatures only. In this paper, we choose VirusShare as our sample source. It is a repository of malware samples, usually adopted in code-signing research [9] [11]. VirusShare samples are named with the form *VirusShare_md5val*, where *md5val* is the sample's MD5 value.We download more than 26 million compressed samples from the VirusShare website. Then we decompress them as the original inputs.

To express the sample size and workload in our work, we show statictics of sample size from related studies in Table III. Our work deal with much more samples than others [9] [8] [11].

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Overview on Verification Tools

Tool \ Item	signature support	show details	how to verify
Sigcheck	Authenticode, Catalog	signer, timestamp, certificate chain, reason for invalid signature	invoke Windows API (e.g. *func WinVerifyTrust* from *wintrust.h*)
Signtool	Authenticode, Catalog	signer, timestamp, certificate chain	
AnalyzePESig	Authenticode, Catalog	signer, timestamp, certificate chain, reason for invalid signature	
Osslsigncode	Only Authenticode	signer, timestamp, certificate chain, reason for invalid signature	open source library (OpenSSL, cURL, etc.)

TABLE II: Data Naming in Pipeline

Data	Naming
Sample Binary	*Virusshare_md5val*
Output Files	*Virusshare_md5val_Sigcheck.txt*, *Virusshare_md5val_Signtool.txt*, *Virusshare_md5val_AnalyzePESig.txt*, *Virusshare_md5val_Osslsigncode.txt*
Sequence Strings	*Sequence_Sigcheck.txt*, *Sequence_Signtool.txt*, *Sequence_AnalyzePESig.txt*, *Sequence_Osslsigncode.txt*
Extracted Information	*Virusshare_md5val_extract_Sigcheck.json*, *Virusshare_md5val_extract_Signtool.json*, *Virusshare_md5val_extract_AnalyzePESig.json*, *Virusshare_md5val_extract_Osslsigncode.json*

Algorithm 1: Two-Step Verifications

```
Input: path ← file path of a sample
Output: outputs ← verification output files
res_1 ← verify_with_Sigcheck (path);
res_2 ← verify_with_Signtool (path);
res_3 ← verify_with_AnalyzePESig (path);
status_1 ← read_status_Sigcheck (res_1);
status_2 ← read_status_Signtool (res_2);
status_3 ← read_status_AnalyzePESig (res_3);
if status_1, status_2, status_3 = not signed then
  | outputs ← None;
else
  | res_4 ← verify_with_Osslsigncode (path);
  | outputs ← {res_1, res_2, res_3, res_4};
end
return outputs;
```

2) Two-Step Verifications: With VirusShare samples, we already acquire enough samples to launch the verification procedures. However VirusShare provides samples including multiple file types, not specialized for code-signing purposes. If we start the verification process directly, quite a few irrelevant outputs will be generated and need efforts to be excluded. We design a two-step verification process to deal with samples, discarding not signed samples in the first step. Procedures are described in Algorithm 1.We divide four verification tools into two groups: a) *Sigcheck*, *Signtool*, and *AnalyzePESig* are adopted in the first step, while b) *Osslsigncode* is adopted in the second step. If tools in the first step all assert a sample to be not signed, the sample will be abandoned without participating in the second step. Meanwhile, for every signed sample, we save its four output files separately. Output files are named *VirusShare_md5val_toolname.txt*, such as *VirusShare_0003c4026a2110716276e1c5889bdc3d_Sigcheck.txt*.

All the verifications are conducted in a cloud service from Alibaba Cloud, with Windows Server 2016 on two cores vCPU and 8 GB RAM. The verification python script is launched with a multi-thread mode. Before the two-step verifications start, some tips need to be noticed: 1) keep the system update to be the latest. 2) configure the system policy for code-signing correctly [16]. 3) be cautious to avoid encoding faults when saving output files. 4) all scripts should be as automatic as possible. 5) elaborately design the directory structures and naming methods in the methodology, to deal with a number of samples.

Two-Step Method: The two-step verifications aim to exclude unwanted samples in advance as many as possible. It efficiently excludes 78.8% samples which do not contain code-signing signatures. After that, we keep 5.5 million samples left. Step-two tools (only *Osslsigncode* now) can directly verify these filtered samples. More tools we inspect, more efficient the verification procedure becomes.

We arrange three tools which invoke Windows API in the first step. They are enough to exclude not-signed samples. Because if a sample is asserted to be not signed by first-step tools (Windows API), it will also be identified as not signed by the Windows system in all probability. Defence mechanisms for code-signing such as UAC (User Account Control) [26] or SmartScreen [24] become useless for these samples. It is useless to take them into account ever. On the other hand, as long as there is a well formatted PKCS#7 signed data structure exists in the sample, step-one tools will assert the sample to be signed (signature status may be invalid). And there can be some malicious samples that just copy signatures from other files (e.g. using *SigThief* [28] to bypass anti-virus programs). These samples' signatures are definitely invalid and meaningless, but samples are not excluded during the two-step verifications.

No Third-Party Filter: Comparatively speaking, we do not use some popular third-party toolkits such as PEfile [1] and

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Sample Size in Related Research

Item / Issue	sample source	sample size
Certify Potentially Unwanted Programs (PUP) [9]	CCSS, VirusShare, NetCrypt, Malicia, Italian	356 thousand
Code-Signing Certificate Revocation [8]	WINE, VirusTotal, Symantec, etc.	965 thousand
Code-Signing Certificate in Korea [11]	VirusShare	5.9 million
Tool Inspection (this work)	VirusShare	26 million

TrID [29] to recognize and exclude non-PE files in our pipeline because of compatibility concerns. Some malformed PE files are too complicated to be recognized by these toolkits, especially for malware developments. The compatibility problem was also explained in another research [11].

B. Stage 2: Outputs Studies

In stage two, we begin to learn these outputs. To precisely extract related fields, we have to fully understand the output format for each tool. We separately prepare four python scripts (named *gather_toolname.py*, e.g. *gather_Sigcheck.py*) to read different tools' outputs, gathering structure information as well as program-execute exceptions.

In most cases, output files can be recognized in the form of *key-value*. So we design a universal method in scripts to learn output structures, assisted by manual work. For each tool, one *gather_toolname.py* script reads corresponding output files and extracts the *key*'s name in each text line, and then joints these *key* names to constitute a sequence string. The sequence string form can be abstracted as follows:

$$\{Key1\|......\|KeyN\}$$

We prepare an empty list in every *gather_toolname.py* script to save possible sequence strings. When processing with an output file, if the captured sequence string does not exist in the current list, we add it to the list. We finally acquire four sequence string lists (named *Sequence_toolname.txt*, e.g. *Sequence_Sigcheck.txt*). Although each list we finally obtain may does not cover all possibilities of a tool, it is enough for us to extract interested information from existing output files. We display a sequence string excerpt example from *Sigcheck* below.

$$\begin{aligned}&\{Signature\ Status\|Signing\ date\|Signer^{*}\|\\&Cert\ Status\|Cert\ Issuer\|SerialNumber\|\\&Thumbprint\|Valid\ from\|Valid\ to\}\end{aligned} \quad (1)$$

With sequence string lists, we are able to summarize regular expressions for these outputs. We present the regular expression for *Signtool* outputs in Section IV-B, Figure 3.

Manual Work: Reading each text line from an output file and constituting a sequence string is a ideal description for *gather_toolname.py* scripts. However, the reality is that *gather_toolname.py* script-writing needs to be manually iterated many times before the final version, which is a repetitive process.

Manual work occurs because: *a)* there are some text lines only contain the *value* part. For example, a text line that represents signer's identifier may be 'Alibaba', not 'Signer: Alibaba'. For these lines, we need to infer their *key* names by context and then create uniform *key* names (e.g. we rename specific signer identifiers as *Signer* in the example (1) above, marked with *). When writing scripts, we need extra efforts to locate these lines and rename *key* names.

b) Similar to condition *a)*, some sequence strings contain specific values. These values may be variable and are unhelpful for summarizing output structures. We need to find out them and replace specific values with placeholders.

c) Output files with program-execute exceptions do not satisfy the *key-value* form. And we do not know in advance possible exception descriptions that appear in output files. If we just mark all output files that we cannot recognize as 'exception', but leave no details, this is not kind for later analyses. We can only collect exception descriptions by manual manners.

At the very start, our *gather_toolname.py* scripts will acquire a great deal of sequence strings but most of sequences are redundant. As we pay manual work in script-writing, numbers of sequence strings become smaller and smaller. Finally we get 383 kinds of sequence strings.

Summary of Output Structures: With sequence strings and regular expressions, we have a clearer vision at output structures. We summarize verification tools' output details in Table IV. Different details in tools are:

Limited[1]: If there are multiple signatures in a sample, *Sigcheck* and *AnalyzePESig* will print 'valid' as long as any signature in the sample is valid. *Signtool* will print *'Successfully Verified.'* only if all signatures are valid. But *Signtool* will not figure out which signature is invalid when *'Successfully Verified.'* is not present. *Osslsigncode* will separately print every signature's status.

Limited[2]: AnalyzePESig shows signing time of timestamp, but does not show the certificate chain of timestamp signer.

Limited[3]: If there are multiple signatures in a sample, *Sigcheck*, *Signtool*, and *Osslsigncode* will print details (signer and cert-chain, timestamp and cert-chain) for each signature. But if not all signatures are valid, *Sigcheck* will only print information of valid ones.

C. Stage 3: Information Extracting and Comparing

In stage three, with acquired sequence strings and regular expressions, we can locate and extract critical code-signing information from output files. These slices are what we really care about in signature segments. Then, we compare these extracted fragments with each other.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Verification Tools Output Details

Item \ Tool	*Sigcheck*	*Signtool*	*AnalyzePESig*	*Osslsingcode*
signature status	Yes (limited[1])	Yes (limited[1])	Yes (limited[1])	Yes
reasons for invalid	Yes	No	Yes	Yes
signer and cert-chain	Yes	Yes	Yes	Yes
timestamp and cert-chain	Yes	Yes	Yes (limited[2])	Yes
multi-signature info	Yes (limited[3])	Yes	No	Yes

We determine interested information to be extracted as follows. An extracted information example is shown in Section IV-C, Figure 4.

- **signature status**: the result of verification, such as valid, invalid, not signed, known error descriptions, or unknown error. If the status is invalid, we attach specific reasons in this item. If there are multiple signature that exist in a sample, we assert the status as valid if any of signatures is valid.
- **signature count**: how many signatures exist in the sample.
- **timestamp count**: how many timestamps exist in the sample.
- **signature information**: a set of signer and timestamp details, for single or multiple signatures.
  - **code-signing signer**: identifier of the code-signing signer, such as signing certificate thumbprint.
  - **timestamp**: exact time marked in the timestamp.
  - **timestamp signer**: identifier of the timestamp signer.

We prepare four scripts (named *extract_toolname.py*, e.g. *extract_Sigcheck.py*) to extract information from outputs. And we save extracted information as json files (named *VirusShare_md5val_extract_toolname.json*, e.g. *VirusShare_md5val_extract_Sigcheck.json*). We organize four json files that belongs to the same sample as a set. Then, we deal with each set using a *compare.py* script, trying to find inconsistent conditions between verification tools.

Extraction: When extracting information, some fields cannot be directly acquired, but need to be inferred by context, such as *signature count* and *timestamp count*. As we have known all structures for outputs from stage two, we can infer this information with no doubt. Furthermore, for exception conditions, we can use exception descriptions obtained by *gather_toolname.py* scripts, placing them in the 'signature status' item.

Comparison: When we compare extracted information from different tools, we assert they are inconsistent if any extracted item is different from other tools'. This assertion is coarse-grained and will cause some false positive conditions, because not all tools provide the whole items above. And different tools may print details with different extents, making some inconsistent conditions are reasonable. We show deeper analyses in Section V.

IV. Data Collection

A. Verification Output Files and Filtered Dataset(via Stage 1)

We download more than 26 million Virusshare samples. It takes 13.42 TB for data storage. After the first verification step, 3.16 TB (about 5.5 million) samples are left. The two-step verification design excludes about 78.8% of original samples.

Remaining samples (5.5 million) are processed with all four verification tools. And finally, we get 70.7 GB (22 million) output files. All output files are saved with the name *VirusShare_md5val_toolname.txt*, such as *VirusShare_0003c 4026a2110716276e1c5889bdc3d_Sigcheck.txt*.

After Stage 1, we acquire a dataset of signed samples. We save this dataset. If we want to add new tools in our pipeline and inspect them, we can use new tools to directly verify this dataset to get new tools' output files. Meanwhile, we publish metadata (file name, signer identifier, signer certificate thumbprint) of samples with valid signatures in our Github webpage. Any researcher can: 1) follow this list to download specific signed samples from VirusShare; or 2) refer to MD5 values appear in file names, to search samples' analysis reports from online malware-analysis platform, such as VirusTotal.

B. Output Structures (via Stage 2)

1) Sequence Strings: With *gather_toolname.py* scripts, we gather 61 kinds of sequence strings for *Sigcheck*; 188 for *Signtool*; 134 for *Osslsigncode*. *AnalyzePESig* output structures are known by reading its source code, but we still run scripts to gather its exception conditions.

We finally gather multiple sequence strings for each single tool. Reasons for this variety are: a) Some irrelevant structure information is also gathered, the redundant items are unavoidable during gathering, because we cannot predict and exclude them in advance. b) Samples may contain multiple signatures and timestamps, bringing about variety in output structures. c) We find that the verification process is interrupted sometimes when *Sigcheck* analyzes specific samples, leaving the output structures malformed.

2) Regular Expressions: With the sequence strings we gathered, we can summarize regular expressions for verification tool outputs. A simplified regular expression for *Signtool* outputs is displayed in Figure 3. Irrelevant items for code-signing are not presented. *Sigcheck*, *AnalyzePESig*, and *Osslsigncode* hold similar quantity of details.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

```
{
  (
    Unable to verify this file using a
        catalog.
    (
        Signature Index: (.*)
        Hash of file: (.*)
        Signing Certificate Chain:
        (
            Issued to
            Issued by
            Expires
            SHA1 hash
        )+
        (
            (
                The signature is
                    timestamped: (.*)
                Timestamp Verified by:
                (
                        Issued to
                        Issued by
                        Expires
                        SHA. hash
                )+
            )|
            (
                File is not timestamped.
            )
        )
    )+
  )|
  (
    File is signed in catalog:
    Hash of file: (.*)
    Signing Certificate Chain:
    (
        Issued to
        Issued by
        Expires
        SHA1 hash
    )+
    (
        (
            The signature is timestamped:
                (.*)
            Timestamp Verified by:
            (
                    Issued to
                    Issued by
                    Expires
                    SHA. hash
            )+
        )|
        (
            File is not timestamped.
        )
    )
  )
  (Successfully verified.)*
  Number of signatures successfully
     Verified: (.*)
  Number of warnings: (.*)
  Number of errors: (.*)
}
```

Fig. 3: Simplified Regular Expression (Signtool)

3) Program-Execute Exceptions: With *gather_toolname.py* scripts, we recognize some output files as program-execute exceptions. *Sigcheck* fails to verify samples sometimes. It prints words such as 'Error while attempting to process file: Invalid access to memory location.' or 'Error while attempting to process file: Error 0xe06d7363'. *AnalyzePESig* may fail to recognize samples with malformed PE formats. It will print words such as 'Read the wrong number of bytes', 'Error no IMAGE_NT_SIGNATURE', or 'Error no IMAGE_DOS_SIGNATURE'. *Osslsigncode* also fails and prints 'Unrecognized file type', 'Failed to calculate DigitalSignature', 'Read stream data error', 'Failed to extract PKCS7 data', and so on. In this respect, *Signtool* performs well.


```
{
  "signature status": "valid",
  "signature count": 2,
  "timestamp count": 2,
  "signature info": [
    {
      "code-signing signer": "0
         CCE41B66788C3A0E4C0E3
         F2C05FCAF571A013C6",
      "timestamp": "Fri Dec 28 18:04:04
         2018",
      "timestamp signer": "03
         A5B14663EB12023091B84
         A6D6A68BC871DE66B"
    },
    {
      "code-signing signer": "8066
         DB916D4F003858643B
         EDD43B5264485734D0",
      "timestamp": "Fri Dec 28 18:04:07
         2018",
      "timestamp signer": "36527
         D4FA26A68F9EB4596
         F1D99ABB2C0EA76DFA"
    }
  ]
}
```


Fig. 4: Extracted Information Example (Signtool)

C. Extracted Information (via Stage 3)

An extracted information example of *Signtool* is shown in Figure 4 (from *VirusShare_0003c4026a2110716276e1c5889-bdc3d_extract_Signtool.json*). *Sigcheck* and *Osslsigncode* get equivalent information, while *AnalyzePESig* is designed to only print the first valid signature it meets.

V. Analysis and Discussion

In an ideal circumstance, the extracted information should be consistent with each other. However, by comparing extracted fragments, we find some inconsistent cases. We present three types of cases in this Section. And then we discuss the common point appears in two cases: their verifying is broken. Finally, we talk about our future works.

A. Case 1: Verifying Neglect

By comparing extracted information, we find a logic flaw in *Sigcheck*: Catalog signatures will be neglected while *Sigcheck* is designed to verify Authenticode signatures first. Then anyone can append an invalid Authenticode signature in a Catalog-signed sample, and *Sigcheck* only displays that the tampered sample contains an Authenticode signature and it is invalid.

We find that the signer of *VirusShare_e2585e4c24690cb418 142c80439b1e5d*[4] is inconsistent according to different tools'

[4]https://www.virustotal.com/gui/file/e2585e4c24690cb418142c80439b1e5d

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

outputs. *Sigcheck* and *Osslsigncode* show the sample is signed by 'F-Secure Corporation'. However, *Signtool* and *AnalyzePESig* show it is signed by 'Microsoft Windows'. We manually check the sample. We find it contains a Catalog signature and an Authenticode signature simultaneously. Its Catalog signature is signed by 'Microsoft Windows', but Authenticode signature is signed by 'F-Secure Corporation'. Actually, the sample is a system file and should only contain the Catalog signature. Its Authenticode signature is abused.

Things become clear. There is a practical situation that a sample contains Catalog signatures and Authenticode signatures simultaneously. *Sigcheck* will search and verify Authenticode signatures first, while *Signtool* and *AnalyzePESig* deal with Catalog signatures first. As a result, they show inconsistent signatures details.

We think it is a logic flaw to verify Authetnicode signatures first. Because Catalog signatures are usually signed by Microsoft for Windows system files. And when verifying Catalog signatures, the verifier will search specific system directories (e.g., *%SystemRoot%\System32\CatRoot*) to find signed Catalog files (.cat). In this sense, it is more credible to verify Catalog signatures.

We investigate its impact. First of all, containing both kinds of signatures is not forbidden in code-signing specifications. Verifications for these two kinds of signatures do not affect each other. And we factually find that some samples contain two kinds of signatures, which are all signed by Microsoft. On the other hand, we totally capture four certificates from three companies which sign extra Authenticode signatures in this condition. They all sign the same system file *'msvcr100_clr0400.dll'* and the Authenticode signatures they created are valid according to *Sigcheck* outputs. We search these tampered samples in VirusTotal, it shows they are not malicious. We contact the certificate owners, one of them explains this is caused by inadvertence, and containing both kinds of signatures is not rare in software development. At last, the VirusTotal platform which integrates *Sigcheck* to analyze signatures is affected. Platform users and related research [10] [7] [11] [8] will get incorrect signature information.

We evaluate its risks. Any localhost computer which deploys *Sigcheck* as a baseline to verify signatures will get incorrect signature information. More seriously, anyone can append an invalid Authenticode signature in a Catalog-signed sample, to hide its valid Catalog signatures. *Sigcheck* will only show the sample contains an invalid Authenticode signature, without any Catalog signature information. Terminal users will be confused by this result.

We alert related developers. We post a discussion about *Sigcheck* at Microsoft Q&A webpages (*Sigcheck* is maintained by Microsoft now) and offer two suggestions for *Sigcheck*: 1) verify Catalog signatures first, like *Signtool* and *AnalyzePESig*. 2) provide a parameter to enable users to consciously choose which kind of signatures (Authenticode or Catalog) to verify. We have not gotten reply from *Sigcheck* developers. We also contact VirusTotal website. Its official says VirusTotal can do nothing unless *Sigcheck* fixes this flaw.

B. Case 2: Timestamp Disturbance

By comparing extracted information, we find a timestamp-verification difference: tools that invoke Windows API process a more strict strategy for timestamp. They directly assert a signature to be invalid if it contains invalid timestamp.

We find that Windows API (*Sigcheck*, *Signtool*, and *AnalyzePESig*) asserts the Authenticode signature of *VirusShare_0d1450606e1e0ea9242fcad86dc8c6b7*[5] as invalid. The reason for the invalid signature is: 'The timestamp signature and/or certificate could not be verified or is malformed'. However, *Osslsigncode* shows its timestamp is invalid, but the code-signing signature is asserted as valid.

We observe the sample binary. It contains an Authenticode signature with timestamp. The timestamp is signed after the timestamp certificate expires. Undoubtedly, its timestamp signature is fake and invalid. Then we observe its code-signing certificate. We find the certificate is still within its validity period, not revoked. Therefore, we think its code-signing signature (without timestamp) should be valid. To test this, we manually modify the sample. We delete the timestamp part from the PKCS#7 structure of the signature, as illustrated in Figure 1. Then we verify the modified sample again, and tools that invoke Windows API assert the code-signing signature to be valid. This means Windows API processes a more strict strategy for timestamp. Anyone can replace valid timestamp with invalid one, and then invalid timestamp will disturb the verification of the code-signing signature. We report this case to Microsoft to ask for further details. Microsoft research just explains that it is by design. Furthermore, this also alerts software developers that they should verify the timestamp before putting it into a code-signing signature.

C. Case 3: Compatibility and Robustness Issues

We also find some compatibility and robustness problems.

1) Compatibility:

Some samples contain complicated PE structures. Thus non-official tools (*Sigcheck*, *AnalyzePESig*, and *Osslsigncode*) may fail to parse samples' signature structures. For example, *Sigcheck* may print 'Error while attempting to process file: Invalid access to memory location'. *AnalyzePESig* may print 'Error no IMAGE_NT_SIGNATURE'. And *Osslsigncode* may print 'Unrecognized file type'. *Signtool* performes much better.

We find that *Osslsigncode* asserts some signatures as valid, while Windows API prints: 'The digital signature of the object is malformed. For technical detail, see security bulletin MS13-098'. The security bulletin MS13-098 [17] [18] describes an opt-in update, which makes Authenticode signature verification more strict to defend against remote code execution. There is no much information about the update disclosed to public. So it is hard for *Osslsigncode* to support this feature.

2) Robustness:

We notice that *Sigcheck* prints incomplete or incorrect information sometimes. We also post a discussion at Microsoft webpages.

[5]https://www.virustotal.com/gui/file/0d1450606e1e0ea9242fcad86dc8c6b7

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

- Bug 1: incomplete information. When there are multiple Authenticode signatures in a PE file, *Sigcheck* may show incomplete signature information. For example, We capture a sample contains three signatures (No.1 is valid, No.2 is invalid, No.3 is valid). When *Sigcheck* verifies the sample, it only shows the first signature information. Because once it verifies an invalid signature (No.2), it will stop, making the result incomplete. This incomplete information is so misleading. Users may think this sample only contains one signature and it is valid.
- Bug 2: incorrect information. When verifying specific samples, *Sigcheck* incorrectly shows the same signature information twice. More seriously, this problem can occur with bug 1 concurrently. For example, a sample contains two Authenticode signatures (No.1 is valid, No.2 is invalid). When *Sigcheck* verifies the sample, due to Bug 1, it only shows the first signature information. And for specific samples, it shows No.1 signature information twice. This condition is also misleading. Users will think this sample contains two signatures and they are all valid.

D. Discussion and Future Work

1) Case 1 and 2 in Common: The Broken Verifying

We notice that case 1 and 2 have something in common. They all return the result ('invalid') half way, before they verify decisive code-signing signatures ('valid' indeed). For case 1, before verifying Catalog signatures, *Sigcheck* searches and verifies Authenticode signatures first, then it returns Authenticode signature status as the result. For case 2, before verifying the code-signing signature, Windows API asserts the timestamp is invalid. Then Windows API asserts the code-signing signature to be invalid as well. For these two cases, software developers generate valid code-signing signatures for their software, but they are unexpectedly asserted as invalid. Any malicious attacker can put external data (invalid Authenticode signature or timestamp) to break the verification process.

2) Future Work: We analyze some typical inconsistent cases in this paper. However, our observation towards comparison results is coarse-grained. It is inefficient to manually analyze comparison results before classifying them. In the future, we plan to design a fine-grained comparison method to classify and then analyze comparison results. Besides, some tools assert signatures to be invalid but not giving enough explanations. Then we cannot find out specific reasons for inconsistent cases just by comparing verification outputs we generate. In this condition, corresponding samples should be analyzed with in-depth checking. Furthermore, we plan to elaborately craft some test samples, trying to induce verification tools to return incorrect results.

VI. Related Works

Windows code-signing is designed to protect software from being tampered with, as well as provide non-repudiation properties. However, malicious developers could make use of this technology to earn trust and compromise the system. The famous Stuxnet malware [5], with valid signatures from famous software providers, was found to compromise industrial control systems on specific targets. It has been well studied [6] [30] and some schemes [33] [32] are proposed to detect such attacks. Similarly, attackers during the SolarWinds incident [4] injected malicious code into a SolarWinds DLL file to launch supply chain attacks. Related studies [31] [27] explain its complexity and put forward mitigation recommendations.

Attackers always try their best to obtain valid code-signing certificates to sign malware. Underground trade in code signing certificates [10] is studied, and it presents the first in-depth analysis of this underground trade. Abused certificates to sign potentially unwanted programs (PUP) or malware are studied [9] [7]. Many certificates which were found to sign PUP or malware still keep valid, which results in continuous influences. Abused code-signing certificates in South Korea were collected, filtered, and evaluated in [11]. And the study [8] analyzes further influences of revocation problems.

Microsoft publishes an official code-signing best practices guide [14] to explain implementation details that should be concerned in developments. Related points in code-signing, such as analyses for specific scenarios, are noted in the guide. NIST also publishes a white paper [2] to describe security considerations for code-signing. It defines some code signing use cases and identifies some security problems, then provides some recommendations.

VII. Acknowledgment

This work was supported in part by National Key Research and Development Program of China (Grant No. 2022YFB3903900), the National Natural Science Foundation of China under Grant 62002011, Youth Top Talent Support Program of Beihang University under Grant YWF-22-L-1272.

We are grateful that *VirusShare* provides original samples.

VIII. Conclusion

Code-signing technologies have become more and more important for modern software distributions. Attackers are willing to conceal their malicious code with valid signatures to bypass system checking. In this work, we raise a universal pipeline to inspect Windows code-signing verification tools. We use four representative tools to verify massive samples and learn their outputs. Then, we extract interested code-signing fragments from outputs and compare them to check that if all tools print consistent results and details. We present three types of inconsistent cases: verifying neglect, timestamp disturbance, and compatibility/robustness issues. In some way, our analysis demonstrates the complexity of Windows code-signing signature verification and our methodology raises a universal pipeline to understand and compare different verification tools.

References

[1] E. Carrera *et al.*, "pefile," 2023, https://github.com/erocarrera/pefile.

[2] David Cooper, Andrew Regenscheid, Murugiah Souppaya, "Security Considerations for Code Signing," 2018, https://www.nist.gov/publications/security-considerations-code-signing.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.

[3] Didier Stevens, "Authenticode Tools," 2022, https://blog.didierstevens.com/programs/authenticode-tools.

[4] FireEye, "Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor," 2020, https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.

[5] Joshua Alvarez, "Stuxnet: The world's first cyber weapon," 2015, https://cisac.fsi.stanford.edu/news/stuxnet.

[6] B. Kim and S. Lee, "Conceptual framework for understanding security requirements: A preliminary study on stuxnet," in *Requirements Engineering in the Big Data Era - Second Asia Pacific Symposium, APRES 2015, Wuhan, China, October 18-20, 2015, Proceedings*, vol. 558, 2015, pp. 135–146.

[7] D. Kim, B. J. Kwon, and T. Dumitraş, "Certified malware: Measuring breaches of trust in the windows code-signing pki," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, p. 1435–1448.

[8] D. Kim, B. J. Kwon, K. Kozák, C. Gates, and T. Dumitras, "The broken shield: Measuring revocation effectiveness in the windows code-signing PKI," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, 2018, pp. 851–868.

[9] P. Kotzias, S. Matic, R. Rivera, and J. Caballero, "Certified pup: Abuse in authenticode code signing," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, p. 465–478.

[10] K. Kozák, B. J. Kwon, D. Kim, and T. Dumitraş, "Issued for abuse: Measuring the underground trade in code signing certificate," 2018.

[11] B. J. Kwon, S. Hong, Y. Jeon, and D. Kim, "Certified malware in south korea: A localized study of breaches of trust in code-signing PKI ecosystem," in *Information and Communications Security - 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I*, vol. 12918, 2021, pp. 59–77.

[12] Mark Russinovich, "Sigcheck v2.82," 2022, https://learn.microsoft.com/en-us/sysinternals/downloads/sigcheck.

[13] MichałTrojnara, "osslsigncode," 2022, https://github.com/mtrojnar/osslsigncode.

[14] Microsoft, "Code-Signing Best Practices," 2007, https://learn.microsoft.com/en-us/previous-versions/windows/hardware/design/dn653556(v=vs.85).

[15] Microsoft, "Windows Authenticode Portable Executable Signature Format," 2008, https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx.

[16] Microsoft, "Manage Revocation Checking Policy," 2009, https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc753863(v=ws.11).

[17] Microsoft, "Microsoft Security Advisory 2915720," 2014, https://learn.microsoft.com/en-us/security-updates/SecurityAdvisories/2014/2915720.

[18] Microsoft, "Microsoft Security Bulletin MS13-098 - Critical," 2014, https://learn.microsoft.com/en-us/security-updates/Securitybulletins/2013/ms13-098.

[19] Microsoft, "Introduction to Code Signing," 2017, https://msdn.microsoft.com/enus/library/ms537361.aspx.

[20] Microsoft, "Authenticode Digital Signatures," 2021, https://learn.microsoft.com/en-us/windows-hardware/drivers/install/authenticode.

[21] Microsoft, "Catalog Files and Digital Signatures," 2022, https://learn.microsoft.com/en-us/windows-hardware/drivers/install/catalog-files.

[22] Microsoft, "SignTool.exe (Sign Tool)," 2022, https://docs.microsoft.com/en-us/dotnet/framework/tools/signtool-exe.

[23] Microsoft, "Time Stamping Authenticode Signatures," 2022, https://learn.microsoft.com/en-us/windows/win32/seccrypto/time-stamping-authenticode-signatures.

[24] Microsoft, "Microsoft Defender SmartScreen," 2023, https://learn.microsoft.com/en-us/windows/security/operating-system-security/virus-and-threat-protection/microsoft-defender-smartscreen/.

[25] Microsoft, "PE Format," 2023, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format.

[26] Microsoft, "User Account Control overview," 2023, https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/.

[27] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. B. Michael, "Perspectives on the solarwinds incident," *IEEE Security & Privacy*, vol. 19, no. 2, pp. 7–13, 2021.

[28] J. Pitts, "SigThief," 2015, https://github.com/secretsquirrel/SigThief.

[29] M. Pontello, "TrID," 2017, https://www.mark0.net/soft-trid-e.html.

[30] M. A. Z. Raja, H. Naz, M. Shoaib, and A. Mehmood, "Design of backpropagated neurocomputing paradigm for stuxnet virus dynamics in control infrastructure," *Neural Comput. Appl.*, vol. 34, no. 7, pp. 5771–5790, 2022.

[31] S. Raponi, M. Caprolu, and R. D. Pietro, "Beyond solarwinds: The systemic risks of critical infrastructures, state of play, future directions," in *Proceedings of the Italian Conference on Cybersecurity, ITASEC 2021, All Digital Event, April 7-9, 2021*, vol. 2940, 2021, pp. 394–405.

[32] T. Siiskonen and M. Rantonen, "Detecting stuxnet-like data integrity attacks," *Secur. Priv.*, vol. 3, no. 5, 2020.

[33] J. Tian, R. Tan, X. Guan, Z. Xu, and T. Liu, "Moving target defense approach to detecting stuxnet-like attacks," *IEEE Trans. Smart Grid*, vol. 11, no. 1, pp. 291–300, 2020.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on July 11,2024 at 08:22:38 UTC from IEEE Xplore. Restrictions apply.