DNSCHK: Malicious Download Detection with Highly-Available Distributed Systems

Anonymous Author(s)

Abstract*—Downloading resources over the internet comes with many risks, including the chance that a malicious actor has replaced the resource you think you are accessing with a compromised version. The current standard for addressing this risk is the use of *checksums* coupled with a secure transport layer; users download a resource and compare its checksum with a posted checksum from the developers to ensure a match. Among the many problems with the current use of checksums are (1) user apathy—for most users, hand-calculating the checksum and comparing to the published version is too tedious; and (2) co-hosting resources and checksums—a malicious actor who compromises a resource can usually trivially compromise a checksum hosted on the same system. In this paper, we propose DNSCHK, a novel resource validation scheme meant as a complete replacement for current checksum based approaches. DNSCHK automates the tedious parts of verification to eliminate user apathy while leveraging authenticated highly-available distributed systems to separate resources from checksums, making these systems and their end users more resilient to resource integrity attacks by avoiding co-hosting. We carefully evaluate the security, performance, and practicality of DNSCHK through implementing extensions in both a web browser (Chrome) and FTP client (FileZilla); implementations are tested versus common resource integrity violations. We find that DNSCHK is more effective than existing solutions, detects a wide variety of real-world integrity errors across a diverse set of platforms, and is a scalable and immediately deployable solution.

I. Introduction

In 2010, through compromising legitimate applications available on trusted vendor websites, nation-state actors launched the Havex malware, targeting aviation, defense, pharmaceutical, and other companies in Europe and the United States [29, 41]. In 2012, attackers compromised an official phpMyAdmin download mirror hosted by reputable software provider SourceForge. The backdoored version of the popular database frontend was downloaded by hundreds of users, potentially allowing attackers to gain access to private customer data [45, 46]. In 2016, attackers broke into the Linux Mint distribution server and replaced a legitimate Mint ISO with one containing a backdoor, infecting hundreds of machines with the malware [5, 51]. Over a four day period in 2017, users of the popular HandBrake open source video transcoder on Mac/OSX were made aware that, along with their expected video software, they may have also downloaded a trojan that was uploading their sensitive data to a remote server [28]. HandBrake developers recommended users perform checksum validation to determine if their install was compromised [27].

As every internet user is certainly aware, downloading resources over the internet comes with considerable risk. This risk can be divided into three categories: response authentication, communication confidentiality, and resource integrity. Response authentication allows us to determine if a response received indeed originates from its purported source through, for instance, the adoption of a Public Key Infrastructure (PKI) scheme [13]. Communication confidentiality, on the other hand, allows us to keep the data transacted between two or more parties private except to said parties through some form of encryption, such as AES [16]. Finally, resource integrity allows us to verify that the data we are receiving is the data we are expecting to receive.

When it comes to response authentication and communication confidentiality concerns on the internet, the state of the art in attack mitigation is Transport Layer Security (TLS) and its Hyper Text Transfer Protocol (HTTP)/PKI based implementation, HTTPS [10, 13, 19, 20, 47]. Assuming well behaved certificate authorities and modern browsing software, TLS and related protocols, when properly deployed, mitigate myriad attacks on authentication confidentiality.

However, as a *communication* protocol, TLS only guarantees the integrity of each *end to end communication* via message authentication code (MAC) [20]. But protected encrypted communications mean nothing if the contents of those communications are corrupted before the fact. Hence, the integrity of resources at the application layer (rather than the transport layer) is outside of the model addressed by TLS and HTTPS [20, 47].

Attacks on resource integrity can be considered a subset of *Supply Chain Attacks* (SCA). Rather than attack an organization directly, SCAs are the compromise of an organization's software source code (or any product) via cyber attack, insider threat, upstream asset compromise, trusted hardware/vendor compromise, or other attack on one or more phases of the software development life cycle [44]. These attacks are hard to detect, even harder to prevent, and have the goal of infecting and exploiting victims by abusing the trust between consumer and reputable software vendor [42].

Ensuring the integrity of resources exchanged over the internet despite SCAs and other active attacks is a hard and well studied problem [1, 2, 11, 13, 23, 36, 40, 47]. For a long time, the de facto standard for addressing this risk in the generic case is with the use of *checksums* coupled with some secure transport medium like TLS/HTTPS. Checksums in this context are cryptographic digests generated by a cryptographic hashing function run over the resource's file contents. When a user downloads a file from some source, they are expected to run the same cryptographic hashing function over their version of the resource, yield a local checksum, and match it with the authoritative checksum given to them from said source.

However, checksums come up short as a solution to the resource integrity problem. Foremost is a well-understood but harsh reality: *user-apathy*—a non-trivial number of users will not be burdened with manually calculating checksums for the resources they download. While detailing how they gained unauthorized access to the servers, one of the hackers behind the 2016 breach of Linux Mint's distribution system went so far as to comment (in respect to checksums): "Who the [expletive] checks those anyway?" [53]. Hardly unique to checksum calculation, cryptographic schemes from HTTPS to PGP have found user apathy a difficult problem space to navigate [3, 54].

Even if a user felt the urge to manually calculate a checksum, they must search for the corresponding authoritative checksum to verify that calculation. As there is no standard storage or retrieval methods for checksums, they could be stored anywhere, or even in multiple different locations that must then be kept consistent; users are not guaranteed to find an authoritative checksum, even if they are published online somewhere. If they do manage to find the authoritative checksum and also recognize the checksums are different, the user is then expected to "do the right thing," whatever that happens to be in context.

Then there is the futility of *co-hosting* a resource and its checksum on the same distribution system. While cost-effective compared to hosting two or more discrete systems—one for the resource and one for the resource's checksum—an adversary that compromises a single distribution system hosting a resource and its checksum can mutate both, rendering the checksum irrelevant. The co-hosting problem was demonstrated by the 2016 hack of Linux Mint's distribution server [5, 51].

Checksums as they are employed currently are not effective at guaranteeing resource integrity. Recognizing this, some corporations and large organizations rely instead on PKI-based solutions such as digital signature validation and code signing [13]. These solutions have been deployed successfully to mitigate resource integrity attacks in mature software package ecosystems (*e.g.,* Debian/apt, Red Hat/yum, Arch/pacman) and walled-garden app stores like Google Play, Apple App Store, and the Microsoft Store.

Unfortunately, not all resources available on the internet are acquired through software package ecosystems with built-in PKI support, nor are all resources software binaries; moreover, these PKI schemes are not compatible with one another and cannot scale to secure arbitrary resources on the internet without significant cost and effort. Worse, roll-your-own PKI is *hard to get right* [13], implying systems built atop them are inherently more *fragile*—susceptible to malfunction due to small errors or misconfigurations.

In this paper, we propose DNSCHK, a novel method of verifying the integrity of resources downloaded over the internet that is a complete replacement for traditional checksums. We approach the problem with four key concerns in mind: a) implementations provide security guarantees transparently without adding any extra burden on the end user—here, an optimal solution avoids relying on the user to overcome apathy in the interest of security; b) configuring the validation method is simple for service administrators and system operators to integrate and deploy while ensuring configuration of a potentially-expensive discrete secondary system to host checksums is unnecessary; c) the validation method is not tightly coupled with any particular highly-available system; and d) no new HTML or JavaScript language additions, application source changes, or user-facing web server/infrastructure alterations are necessary. We implement DNSCHK as a proof-of-concept Google Chrome extension as well as a patch to the FileZilla FTP client.

We evaluate the security, scalability, and performance of our automated defense against resource corruption to demonstrate the effectiveness and practicality of the DNSCHK approach. Specifically, we find no obstacles to efficient scalability given choice of authenticated distributed system and no performance overhead compared to downloads without DNSCHK. We further provide a publicly accessible empirical demonstration of DNSCHK's protective utility via a patched HotCRP instance[1].

In summary, our primary contributions are:

- We propose a novel practical defense against receiving corrupted or compromised resources over the internet. Contrasted with current solutions, our defense requires no source code or end user facing web server/infrastructure changes in a browser-based or similar environment, does not employ unreliable heuristics, does not interfere with other software or extensions that also handle download security, and can be transparently deployed without adding to the *fragility* of DNSSEC-enabled systems; *i.e.,* , it can protect users whose software implements the DNSCHK approach while remaining unnoticeable to users whose software does not.
- We present our prototype DNSCHK implementations for Google Chrome and FileZilla and demonstrate their effectiveness in automatically and transparently mitigating the accidental consumption of compromised resources from a compromised server hosting a compromised web portal. To the best of our knowledge, this is the *first* system providing such capabilities with little implementation cost while requiring no additional work from the end user.
- We carefully and extensively evaluate the security, scalability, and performance of the DNSCHK approach and provide a publicly accessible empirical demonstration of its protective utility.

We release the DNSCHK solution to the community as open source software to prompt exploration of the DNSCHK approach.[2]

[1]The patched HotCRP instance is available at https://tinyurl.com/dnschk-hotcrp

[2]The DNSCHK Chrome extension is available at https://tinyurl.com/dnschk-actual

II. Background

In this section, we describe the motivation for DNSCHK, including four case studies that frame the threat posed by Supply Chain Attacks (SCAs). We then examine current methods to detect and prevent resource corruption including checksums, HTTPS, anti-malware, PKI, and others.

A. Supply Chain Attacks on Resource Integrity

Modern software development requires a complex globally distributed supply chain and development ecosystem for organizations to design, develop, and deploy, and maintain products efficiently [42]. Such a globally distributed ecosystem necessitates integration with potentially many third party entities be they specialty driver manufacturers, external content distribution network (CDN) providers, third party database management software, etc.

Critically, reliance on third parties, while often cost-effective and feature-rich, also increases the risk of a security compromise at some point in the supply chain [42]. These types of compromises are known as Supply Chain Attacks (SCA). In the context of software development, SCAs are the compromise of an organization's software source code via cyber attack, insider threat, upstream asset compromise, trusted hardware/vendor compromise, or some other attack on one or more phases of the software development life cycle or "supply chain" to infect an unsuspecting end user [44].

Every year, major supply chain attacks become more frequent and their fallout more widely felt [42, 44]. Whether major or minor, SCAs are hard to detect, even harder to prevent, and have the goal of infecting and exploiting victims by violating the trust between consumer and reputable software vendor.

Fig. **??** details the phases of a generic software development supply chain. For the purposes of this research, we focus exclusively on SCAs targeting the deployment, maintenance, and retirement phases.

B. Motivation: Cases

Here we select four historic attacks we believe most effectively articulate the threat posed by resource integrity SCAs and how DNSCHK might have been used to more efficiently mitigate fallout. We examine each attack, noting the critical points of failure in their checksum-based resource security model.

Case 1: PhpMyAdmin. For an unspecified amount of time circa 2012, a compromised download mirror in SourceForge's official HTTPS-protected CDN was distributing a malicious version of the popular database administration software phpMyAdmin [15]. The administrator of the mirror in question confirmed the attack was due to a vulnerability not shared by SourceForge's other mirrors [45].

Attackers mutated the software image, injecting files that would allow any attacker aware of their existence to remotely execute arbitrary PHP code on the victim's system [46]. SourceForge estimates approximately 400 unique users downloaded this corrupted version of phpMyAdmin before the mirror was disconnected from their CDN, potentially allowing attackers access to the private customer data of any number of organizations [45].

While the attackers were able to penetrate a mirror in SourceForge's CDN, the official phpMyAdmin website was entirely unaffected; the authoritative checksums listed on the site's download page were similarly unaffected [45]. Hence, a user who was sufficiently motivated, had sufficient technical knowledge of checksums and how to calculate them, and was also privy to the location of the correct checksum for the official phpMyAdmin image *might* have noticed the discrepancy between the two digests. Clearly, a non-trivial number of users do not meet these criteria, and this attack demonstrates the problem of *user-apathy*.

Case 2: Linux Mint. In 2016, the Linux Mint team discovered an intrusion into their official HTTPS-protected distribution server [51]. Attackers mutated download links originally pointing to the Linux Mint 17.3 Cinnamon edition ISO, redirecting unsuspecting users to a disparate system hosting a custom Mint ISO compiled with the IRC-based Linux backdoor malware *Tsunami* [5]. The attack affected several hundred of the downloads during that day, with the attackers claiming that a "few hundred" Linux Mint installs were explicitly under their control. The primary motivation behind the intrusion was the construction of a botnet [53]. The authoritative checksum displayed on the official website was also mutated to corroborate the backdoored ISO [53], illustrating the *co-hosting* problem.

Storing the checksum elsewhere may have prevented mutations on the checksum; still, as demonstrated by the first case, such an effort is not itself a solution. Hosting a checksum on a secondary system is not very useful if users downloading the resource protected by that checksum are not actually *checking* it against a manual calculation.

Case 3: Havex. As part of a widespread espionage campaign beginning in 2010, Russian Intelligence Services targeted the industrial control systems of numerous aviation, national defense, critical infrastructure, pharmaceutical, petrochemical, and other companies and organizations with the Havex remote access trojan [29, 41]. The attack was carried out in phases whereby innocuous software images hosted on disparate *legitimate* vendor websites were targeted for replacement with versions infected with the Havex malware [41]. The goal here, as is the case with all SCAs, was to infect victims indirectly by having the Havex malware bundled into opaque software dependencies, *i.e.,* a hardware driver or internal communication application.

It is estimated that Havex successfully impacted hundreds or even thousands of corporations and organizations—mostly in United States and Europe [41]. The motivation behind the Havex malware was intelligence exfiltration and espionage [29]. How many of these vendors employed checksums and other mitigations as part of their software

Concept	Design	Development	Integration	Deployment	Maintenance	Retirement
X	X	X	X	✓	✓	✓

TABLE I: The software development supply chain. Attacks outside of the deployment, maintenance, and retirement phases are outside of the DNSCHK threat model; hence, they are not considered.

release cycle is not well reported, though investigators note said vendors' distribution mechanisms were insecure [41]; however, an automated resource validation method could have helped mitigate the delivery of compromised software to end users.

Case 4: HandBrake. In May of 2017, users of HandBrake, a popular open source video transcoder for Mac/OSX, were made aware that they may have downloaded and installed a trojan riding atop their transcoding software. Attackers breached an HTTPS-protected HandBrake download mirror, replacing the legitimate software with a version containing a novel variant of the *Proton* malware [28]. The number of users potentially affected is unreported.

The goal of the attack was the exfiltration of victims' sensitive data, including entire keychains (unlocked), private keys, browser password databases, 1Password/Lastpass vaults, decrypted files, and victims' personal videos and other media [28]. The HandBrake developers recommended users perform manual checksum validation to determine if their installation media was compromised [27].

Despite the attackers mutating the HandBrake binary, the authoritative checksums listed on the official HandBrake download page were reportedly left untouched [27]. Further, the developers of HandBrake store their authoritative checksums both on their official website and in their official GitHub repository [27]. A sufficiently knowledgeable, sufficiently motivated user *might* have noticed the discrepancy between their calculated checksum and the authoritative checksum listed on the download page.

Suppose, however, that the attackers *had* managed to mutate the checksums on the official website. Then there would be a discrepancy between the authoritative checksums on the official site and the authoritative checksums in the GitHub repository—that is *if* users are even aware that a second set of checksums are available at all. On top of technical knowledge, a user in this confusing situation is then expected to "do the right thing," whatever that happens to be in this context.

C. Other Detection and Mitigation Methods

Detecting and/or mitigating resource integrity SCAs and other active attacks is a non-trivial and well studied problem [1, 2, 11, 13, 23, 36, 40, 47]. What follows is a brief overview of current and popular methods to ensure the integrity of resources exchanged over the internet other than checksums.

1) Anti-Malware Software: Anti-malware software are heuristic-based programs designed for the specific purpose of detecting and removing various kinds of malware. However, updates to anti-malware definitions often lag behind or occur in response to the release of crippling malware. For example, during the 2017 compromise of the HandBrake distribution mirror, users who first ran the compromised HandBrake image through *VirusTotal*—a web service that will run a resource through several dozen popular anti-malware products—received a report claiming no infections were detected, despite the empirically verifiable presence of the Proton Malware.

Worse, not all resource compromises end up looking like malware. In the 2012 compromise of SourceForge's CDN, where a malicious version of phpMyAdmin was delivered to hundreds of users, the PHP source itself was altered to enable remote code execution. However, the extraneous code was virtually indistinguishable from the rest of the raw PHP source in the phpmyAdmin image being distributed.

At the time of writing (2018), VirusTotal correctly identifies the compromised version of the Handbrake image as malware.

2) HTTPS / Encrypted Channel: The state of the art in secure communications is the Transport Layer Security (TLS) standard and its Hyper Text Transfer Protocol (HTTP)/PKI based implementation, HTTPS [13, 20, 47]. Assuming well behaved certificate authorities, modern browsing software, and proper deployment, TLS and HTTPS mitigate many authentication and confidentiality related concerns through the establishment of an encrypted authenticated communication channel between parties [20, 47, 48].

HTTPS and related protocols are not a panacea, however. Unlike concerns relating to authenticity and confidentiality, *resource integrity* deals with the content of a communication; specifically: ensuring the bytes received at the end of a transaction are the bytes we expected to receive. For example, a binary expected to be 10MiB, when downloaded over the internet, should not be received as an 11MiB executable—even if the compromised resource remained confidential in transit between authenticated parties. Similarly, it would be ill advised to knowingly execute a 10MiB binary that, for one reason or another, had half its bits flipped by the time it was received. This can occur despite the integrity guarantee provided by TLS/HTTPS because resource integrity attacks and other SCAs fall outside of the threat model addressed by TLS/HTTPS.

Hence, despite the resilient nature of HTTPS and the powerful security properties it guarantees, end users are still vulnerable to attacks on resource integrity. This can occur at the point of distribution, such as a compromised node in a third party CDN that delivers malicious resources masquerading as popular software (cf. cases 1-4). It can occur when an adversary gains control of some aspect of the software deployment process (cf. case 3).

Moreover, an adversary can render HTTPS-protected HTML-based protections such as co-hosted resource checksums, the content-MD5 header [40] and Subresource Integrity

(SRI) [2] digests ineffective by trivially mutating the compromised download portal and/or update/maintenance mechanism.

3) Browser-based Heuristics and Blacklists: Modern browsers employ heuristic and blacklist based detection and prevention schemes in an attempt to protect users from encountering malicious content on the internet. Implementations include Google Chrome's *Safe Browsing* feature, Mozilla Firefox's *Phishing/Download Protection*, and Microsoft Edge's *malware sniffing* Windows Defender browser bundle.

Similar to anti-malware software, browser-based heuristics and blacklists are a reactive rather than proactive solution; hence, they are ineffective at shielding users from attacks on the integrity of the resources downloaded over the internet.

III. THE DNSCHK APPROACH

In this section we detail the DNSCHK approach: a novel defense against receiving corrupted or compromised resources over the internet. We further present the challenges and their solutions in designing DNSCHK that transparently mitigates resource integrity Supply Chain Attacks (SCA) without degrading the experience of users that do not implement the DNSCHK approach.

Further, though our concrete implementations relies on DNS authenticated with DNS Security (DNSSEC), the approach itself is flexible and completely agnostic of any single component. The implementation choice of highly-available distributed *backend*, for instance, is not restricted to the DNS network. The approach works just as well with an authenticated Distributed Hash Table (DHT) or some distributed authenticated key-value store (*e.g.,* Redis) as the backend.

A. Transparency and User Apathy

Human factors such as user apathy have stymied cryptographers for decades. Schemes that are otherwise reasonably cryptographically solid can fail catastrophically due to human error, confusion, or simple lack of interest. Some users are likely to avoid using a security measure altogether if it presents even a minor obstacle to immediate gratification [3, 54]. In the browser, for example, this phenomenon can be observed empirically.

Leveraging the in-browser telemetry of Mozilla Firefox and Google Chrome to passively observe over 25 million warning impressions in 2013, Akhawe et al. found that users of Google Chrome clicked through a quarter of *malware and phishing warnings* and *70% of TLS warnings* [3]. Users also clicked through a third of Mozilla Firefox's TLS warnings and a tenth of their malware and phishing warnings. That is to say: a significant percentage of browser users are *determined* not to let TLS trust issues and/or the threat of malware prevent them from receiving their desired content. Hence, we must assume: some non-trivial number of users, similarly *determined* to transact resources over the internet, will not be burdened with the off-path minutiae of manually calculating a checksum (if they are even familiar with the jargon) and verifying the integrity of the resources they are downloading.

With this assumption in mind, the primary goal of DNSCHK then is to side-step the human factor altogether by providing a completely transparent and unobtrusive, fully-automated method of checksum calculation and verification. We achieve this through 1) the unique identification of individual hosted resources and 2) a globally available mapping of unique resource identifiers to corresponding checksums.

DNSCHK can be imagined as a completely transparent security layer sitting between the user and the resource. Immediately after a resource is downloaded, the DNSCHK layer will generate two cryptographic digests. One digest uniquely fingerprints said resource based on its name. This is known as the *Non-Authoritative Checksum* (NAC) and is yielded from running the cryptographic hashing function over the contents of the resource file. The second digest uniquely fingerprints said resource based on its contents. This is known as the *Resource Identifier* (RI) and is yielded from running the cryptographic hashing function over the resource's public path on the distribution system.

Next, DNSCHK uses the RI to retrieve an *Authoritative Checksum* (AC) from the backend. If successful, DNSCHK will compare the NAC to the AC—we refer to this as *Non-Authoritative Checksum Validation* (NAC Validation). Only in the case where they do not match will the user even realize DNSCHK exists. Otherwise, DNSCHK remains completely transparent the the end user, as demonstrated in our browser-based implementation.

B. Defeating Co-Hosting, Perhaps for Free

Funding and maintaining a single server/system to host all of your assets can be extremely cost-effective in the short term compared to hosting two or more discrete systems—one hosting the resource and one hosting the resource's checksum. Unfortunately, this establishes a single point of failure: an adversary that compromises such a system can both mutate the resource and update the checksum to match the mutation. Hence, *co-hosting* a resource and its corresponding checksum on the same distribution system virtually negates the effectiveness of having a checksum at all. This is widely understood in the security community [5].

Hence, deployment of DNSCHK necessitates the existence of a separate distribution mechanism for resources and ACs. Though the concept of using some distributed authenticated storage service to query a global mapping between RIs and ACs sounds intuitive and straightforward, two natural concerns arise. The first: modern fully authenticated schemes are based on PKI; who is managing this infrastructure and can they be trusted? The second: who is funding the establishment and maintenance of this potentially complex secondary system?

Fortunately, there exists a highly-available fully authenticated globally distributed high performance low latency mapping service that web-facing organizations and IT teams are already quite familiar with (and already pay for): the Domain Name System (DNS). Adding extra resource records to a DNS zone is essentially a costless operation, meaning any organization that already has a DNSSEC-protected web presence can immediately deploy DNSCHK.

To avoid co-hosting in our implementations, we preferred DNS to other candidate highly-available authenticated systems for just these reasons.

C. Platform Diversity

From our evaluation, the computational overhead of running DNSCHK is minimal for most resources. Further, additional network load is negligible (cf. Section IV). Hence, the DNSCHK approach can be incorporated into software on most any device capable of communicating with the chosen backend. This includes desktops, laptops, tablets, mobile devices, embedded systems, etc.

D. Proof-of-Concept Implementations

1) DNSCHK as a Google Chrome Extension: We implement DNSCHK as a proof-of-concept Google Chrome extension that can be configured to work with either global DNS or local DHT as its backend. Our Chrome extension does not make any modifications to the Chrome user interface or viewport (other than the extension icon itself). Further, downloads work exactly the same whether DNSCHK is installed or not—the extension is transparent to end users. If a failure is experienced during NAC Validation, however, DNSCHK will alert the user to the dangerous download via the extension's icon and popup interface.

The extension computes RIs from the full URL path of a resource. For example, considering a web resource hosted at `https://somesite.com/var/downloadme.txt`, our implementation would hash `/var/downloadme.txt` to yield an RI.

Unlike the FileZilla patch implementation, the Chrome extension implementation required a special *Origin Domain (OD)* resolution step based on the Chrome API's `DownloadItem::referrer` and the current Chrome tab's URL. The Origin Domain is the base domain used to query the backend—in this case DNS—and will always be the Second-Level Domain (SLD) fragment of the current tab's URL in our implementation. For example: `somesite.com` would be the OD for the URL `frag.something.somesite.com` and `fakesite.io` would be the OD for the URL `git.fakesite.io`.

The OD is then appended to the Primary Label (PL), which is then appended to the RI Sub-Label (SL). The Primary Label (PL) is a standard string used to identify DNS records that belong to DNSCHK; we used "_dnschk". It will always appear as the third-level domain following the OD in any request to the backend. The RI Sub-Label (SL) is a standard string used to identify DNS records that contain RIs; we used "_ri" in our implementation. The resulting construction, consisting of `SL.PL.OD`, is appended to the RI calculated earlier. This forms the subject of the query to our DNS backend, whereafter the DNS network responds with a TXT record containing the AC or an indication that the RI does not exist in the zone; the latter case is silently ignored.

Note: to remain in compliance with DNS protocol label limits, we split the RI—a 64 character string—into two labels separated by a period. The final construction sent to DNS consists of `RI1.RI2.SL.PL.OD`. This may not be necessary given a different choice of backend.

2) DNSCHK as a FileZilla Source Patch: We implement DNSCHK as a proof-of-concept patch to the FileZilla source. Our implementation does not make any modifications to FileZilla's user interface. As with the Chrome extension, FTP downloads work exactly the same whether a user is using a patched or non-patched version of FileZilla. The extension is similarly transparent to end users. If a failure is experienced during NAC Validation, DNSCHK will alert the user via a popup alert.

The extension computes RIs from the full URI path of a resource on the server (including root). For example, considering a resource hosted on an FTP server at `/ user/var/downloadme.txt`, our implementation would hash `/ user/var/downloadme.txt` to yield an RI.

Unlike the Chrome extension patch implementation, the FileZilla patch requires no special *Origin Domain (OD)* resolution step. The domain to be used for backend queries is the domain of the FTP server itself. The remainder of the implementation comports with the Chrome extension implementation of DNSCHK.

IV. Evaluation

The primary goal of any DNSCHK implementation is to alert end-users when the resource they have downloaded is something other than what they were expecting. We tested the effectiveness of our approach using the DNSCHK extension for Google Chrome, a real-world deployment of HotCRP, and a random sampling of papers published in previous IEEE S&P proceedings.

A. Threat Model and Security Considerations

1) Compromised Resource: We consider the case where an adversary can influence or even completely control the victim's resource distribution mechanism (web page, file server, CDN, etc) in any way. In this context, the adversary can trick the user into downloading a compromised resource of the adversary's choice. This can be accomplished by compromising the resource on the victim's system or tricking the user into downloading a compromised resource on the adversary's remote system.

In this case, the adversary does not have control over any DNS zone(s) relevant to the function of DNSCHK.

If the adversary does not alter the Resource Identifier, the compromised resource will fail integrity validation during the NAC Validation step.

If the adversary does alter the Resource Identifier, there are two possibilities: a) the new Resource Identifier *does not* exist in the DNS zone, in which case DNSCHK will fail to resolve the NAC, hence the NAC Validation step will fail silently; b) the new Resource Identifier *does* exist in the DNS zone, therefore the "new" RI must be pointing to a different resource's checksum. Unless the adversary's goal is to swap one or more resources protected by DNSCHK and a particular DNS zone with another resource also protected by DNSCHK and in that same zone, the NAC Validation step will fail. For the aforementioned "swap" to work, the adversary would be required to both change the RI and also offer to the victim the DNSCHK protected resource the "new" RI corresponds to, which shrinks the attack surface significantly.

We note that, in our proof-of-concept implementation, the "silent failure" above allows DNSCHK to be immediately deployed without risk of annoying the user with false negatives. See Section V for further discussion.

2) Compromised Authoritative Checksum: We consider the case where an adversary can completely control the victim DNS zone(s) that allow DNSCHK to function. Therefore, the adversary can return an authoritative response of their choice to any DNS query.

In this case, the adversary does not have control over the victim's resource distribution mechanism (web page, file server, CDN, etc).

DNSSEC ensures the validity and authenticity of DNS responses. In order for the adversary to control any relevant DNS zones, they must have access to the authoritative DNS server and/or the appropriate DNSSEC keys.

Even if the adversary achieved this level of compromise, they do not have the ability to deliver a malicious payload in this case. However, the adversary could use control over the relevant DNS zones to cause denial-of-service style attacks against those attempting to download the resource by causing all NAC Validation checks to fail. This is mitigated by DNSCHK allowing the user to "override" its error states; *i.e.,* , DNSCHK does not (and cannot, thanks to the Chrome/WebExtensions API) mutate a downloaded resource. See Section V for further discussion on limitations due to the Chrome/WebExtensions API.

3) Compromised Resource and Authoritative Checksum: We consider the case where an adversary can influence or even completely control the victim's resource distribution mechanism (web page, file server, CDN, etc) in any way. Additionally, the adversary can completely control the victim DNS zone(s) that allow DNSCHK to function. Therefore, the adversary can make the user download a compromised resource and also return a (compromised) AC that legally corresponds to said compromised resource.

4) Determining the Origin Domain: If an adversary manages to compromise a web page/server, they have two options. They can mutate the resource directly, which would be observable via DNSCHK. They could also mutate the web/download page itself, replacing the anchor with a malicious one that points to a compromised resource on the adversary's remote system. DNSCHK will still catch this due to the Chrome API's `DownloadItem::referrer` property (distinct from the concept of an HTTP referrer).

However, a clever attacker might be able to trick the Chrome API into populating `DownloadItem::referrer` by redirecting the user to a valid and innocuous page that very quickly redirects the user again to a compromised resource with the goal of tricking the Chrome API into supplying DNSCHK with a chosen `DownloadItem::referrer`.

In order to prevent such implementation-specific attacks, we make a distinction between the domain that the hyperlink containing the desired resource references and the Origin Domain (OD)—or the domain of the document within which said hyperlink exists. The extension is implemented such that the OD is resolved as early as possible in the page loading process. The scope of the OD is at the tab level, meaning there is one OD determined for each open browser tab. Once determined for a tab, the OD should not be recalculated for some period of time. If the tab is navigated and a download is started within a chosen time window, the user will be asked to verify that the OD is what they expect it to be (should be a familiar URL).

We catch potential redirection attacks by assuming any navigation that results in a direct download up to three (3) seconds after the page has loaded is suspicious and requires affirmation by the user. We expect this extreme Chrome-specific edge case to occurrence very rarely, if at all.

B. Real-World Resource Corruption Detection with Google Chrome and HotCRP

To further evaluate the effectiveness of our mitigation, we tested our proof-of-concept DNSCHK Chrome extension implementation against a series of common real-world resource integrity violations. The impetus behind any such resource integrity supply chain attack is to have the resource pass through undetected with the hope that an unsuspecting user will interact with it.

As a test bed, we launched a heavily modified version of the popular open source research submission and peer review software, *HotCRP*. Our modifications allowed us to interactively corrupt submissions and manipulate relevant DNS entries. We uploaded a variety of different resources, including legitimate PDFs, to HotCRP.

We then selectively corrupted half of these resources with random bit flips. Interesting to note: the HotCRP instance did not update the checksums of the resources we corrupted. The only way to notice the attack would be to manually validate PDFs downloaded from HotCRP after the fact, something most users are not willing to do. Further, leveraging our administrator access to the instance, we uploaded files known to be malicious, replacing previously submitted PDFs in the HotCRP backend.

When we attempted to download these resources, DNSCHK flagged each corrupted resource. Unmodified resources were not affected.

Further, we implemented a "redirection" attack wherein a "compromised" PHP script added to the HotCRP instance redirected users several times before quickly triggering the download of a corrupted resource from a disparate domain. DNSCHK correctly flagged this download as suspicious once the download began, successfully warning the user.

We conclude that our proof-of-concept DNSCHK implementation and approach are more effective than existing solutions at detecting integrity violations in arbitrary resources on the internet; this is especially true when DNSCHK is compared to the de facto standard, checksums.

C. Deployment and Scalability

With the HotCRP demo, the totality of our resource deployment scheme consisted of the addition of a new TXT entry to our DNS zone file—accomplished via API call—during HotCRP's paper submission process. This new TXT entry consisted of a mapping between a RI and its corresponding AC.

We find a DNS record addition or update during the resource deployment process to be simple enough for service administrators to implement and presents no significant burden to deployment outside of DNS API integration into an organization's software development supply chain.

As DNSCHK is predicated on a distributed authenticated highly-available backend, we conclude that the scalability of DNSCHK can be reduced to the scalability of its backend. We are aware of no other obstacles to scalability beyond those imposed by the underlying distributed system.

In respect to DNS specifically, packet fragmentation can be a concern for high performance networks [18], but this is an artifact of DNSSEC and related protocols rather than DNSCHK [1]. Further, we are aware of no practical limits or protocol-based restrictions on the scalability of a DNS zone file itself or its subzones. A service can host tens of thousands of resource records in their DNS zone file [38, 39].

D. Performance Overhead

While evaluating DNSCHK, we observed no discernible additional network load or CPU usage with the extension loaded into Chrome; hence, we found that DNSCHK introduces no additional performance overhead outside of requiring Chrome to function. As DNSCHK does not interrupt or manipulate resources as they are being downloaded, there is no additional download latency introduced by DNSCHK.

V. Discussion

In this section, we examine current and previous DNS-based and other cryptographic schemes, most of which are based on public key cryptography. Further, we note PGP's limiting human factors, how those factors also apply to the checksum solution, and how the DNSCHK solution avoids them. Thereafter, we discuss some limitations of the DNSCHK methodology, implementation, and DNS itself.

A. Additional Related Work

Cryptographic Data in DNS Resource Records.

The DNS-Based Authentication of Named Entities (DANE) specification [21, 32, 56] defines the "TLSA" and "OPENPGPKEY" DNS resource records to store cryptographic data. These resource record types, along with "CERT" [34], "IPSECKEY" [49], those defined by DNS Security Extensions (DNSSEC) [1], and others demonstrate that storing useful cryptographic data retrievable through the DNS network is feasible at scale. Due the unique requirements of DNSCHK, however, we use "TXT" records to map Resource Identifiers to Authoritative Checksums. In accordance with RFC 5507 [22], an actual DNSCHK implementation would necessitate the creation of a new DNS resource record type as no current resource record type meets the requirements of DNSCHK.

PGP/OpenPGP. Though PGP addresses a fundamentally different authentication-focused threat model compared with DNSCHK, it is useful to note: many of the same human and UX factors that make the cryptographically solid OpenPGP standard and its various implementations so unpleasant for end users also exist in the context of download integrity verification and checksums. End users cannot and *will not* be burdened with manually verifying a checksum; as was the case with PGP 5.0 [54], some users are likely confused by the very notion and function of a checksum, if they are aware that checksums exist at all. If PGP's adoption issues are any indication, users of a security solution that significantly complicate an otherwise simple task are more likely to bypass said solution rather than be burdened with it. To assume otherwise can have disastrous consequences [54] (also see: Section II).

Link Fingerprints and Subresource Integrity. The Link Fingerprints (LF) draft describes an early HTML anchor and URL based resource integrity verification scheme [36]. Subresource Integrity (SRI) describes a similar production-ready HTML-based scheme designed with CDNs and web assets (rather than generic resources) in mind. Like DNSCHK, both LF and SRI employ cryptographic digests to ensure no changes of any kind have been made to a resource [2]. Unlike DNSCHK, LF and SRI rely on the server that hosts the HTML source to be secure; specifically, the checksums contained in the HTML source must be accurate for these schemes to work. An attacker that has control of the web server can alter the HTML and inject a malicious checksum. With DNSCHK, however, an attacker would also have to compromise the DNS zone or whichever authenticated distributed system hosted the mappings between Resource Identifiers and Authoritative Checksums.

Content-MD5 Header. The Content-MD5 header field is a deprecated email and HTTP header that delivers a checksum similar to those used by Subresource Integrity. It was removed from the HTTP/1.1 specification because of the inconsistent implementation of partial response handling between vendors [23]. Further, the header could be easily stripped off or modified by proxies and other intermediaries [40].

B. Limitations

1) DNSSEC Adoption is Slow: The DNSSEC adoption rate—which has increased dramatically since the first production root zone was signed in 2010 [7, 33]—is decidedly variable and slow to rise. Only around 3% of Fortune 1000 and 9% of university domains have properly deployed DNSSEC [43], and the number of DNS resolvers validating DNSSEC replies currently sits at approximately 12-14% [7].

While this could be happening for a variety of reasons [6, 17, 30, 31, 57], slow growth is certainly not outside of the norm for global protocol deployments that are perceived as "nice to have" rather than "business critical". For instance: the adoption rate of IPv6, proposed nearly 25 years ago, is similarly slow to rise. Globally, when measured as the availability of IPv6 connectivity among users accessing any Google service, it rests at approximately 21% [26], with only 2% of Fortune 1000 and 3% of university domains being IPv6-enabled [43]; we note that this is the case *despite IANA and all RIRs having entered the final stage of virtual IPv4 address space exhaustion as of 2018* [55] while the number of internet-connected devices *continues its upward climb* [12].

Lamentably, as with IPv6, DNSSEC adoption will continue to grow slowly until securing our global DNS infrastructure is more widely considered "business critical."

We also note that, while the highly-available DNS network is used to host the mappings between Resource Identifers and Authoritative Checksums in the DNSCHK implementation, the DNSCHK *approach* is still valid no matter the choice of highly-available distributed system as long as there is an authentication component; *e.g.,* Distributed Hash Table (DHT), distributed authenticated KV store (*e.g.,* Redis), et cetera.

2) DNS-Specific Protocol Limitations: DNS [38] was not originally designed to transport or store relatively large amounts of data, though this has been addressed with EDNS0 [18]. The checksums stored in DNS should not be much longer than 128 bytes or the output of the SHA512 function. Regardless, DNS resource record extensions exist that store much more than 128 bytes of data [32, 34, 49, 56].

Several working groups are considering DNS as a storage medium for checksums/hash output as well, such as securitytxt [24]. A widely deployed example of DNS "TXT" resource records being used this way is SPF and DKIM [14].

Additionally, DNSCHK does not add to the danger of amplification and other reflection attacks on DNS; these are generic DNS issues addressable at other layers of the protocol.

3) Chrome Implementation: Our current JavaScript proof-of-concept implementation, as a Chrome extension, is not allowed to touch the resource file downloaded by Chrome and so cannot prevent the potentially-malicious resource file from being executed by the end user—a feature Chrome/Chromium reserves for its own internal use. The Chrome *app* API [25] might have been of assistance as it allowed for some limited filesystem traversal via a now deprecated native app API; there is also a non-standard HTML5/WebExtensions FileSystem API that would provide similar functionality were it to be widely considered [8].

DNSCHK would be even more effective as a browser extension if Chrome/Chromium or the WebExtensions API allowed for an explicit `onComplete` event hook in the downloads API. This hook would fire immediately before a file download completed and the file became executable, *i.e.,* had its `.crdownload` or `.download` extension removed. The hook would consume a `Promise`/`AsyncFunction` that kept the download in its non-complete state until said `Promise` completed. This protocol would allow DNSCHK's background page to do something like alter the download's `DangerType` property and alert the end user to the dangerous download naturally. These modifications would have the advantage of communicating intent through the browser's familiar UI and preventing the potentially-malicious download from becoming immediately executable. Unfortunately, the closest the Chrome/WebExtensions API comes to allowing `DangerType` mutations is the `acceptDanger` method on the downloads API, but it is not suitable for use with DNSCHK as a background page based extension.

C. Future Work

1) Merkle Trees and Early Resource Validation: Using Merkle trees [37] instead of pure cryptographic hashing functions for resource validation would enable partial verification of large files. For example, suppose we are downloading a 10TiB resource and it is compromised. By calculating a Merkle tree beforehand, we do not have to wait for the resource to finish downloading before we render a failing judgment. This partial verification has the potential to save the user a significant amount of time, though using Merkle trees for resource integrity validation over the internet is decidedly not-trivial [9].

For a production example of Merkle tree based resource integrity validation, we can look to the so-called *Tiger tree hash* [4, 37] (TTH) construction. The TTH, a Merkle tree implementation, is built on the Tiger cryptographic hashing function. Merkle trees and TTHs are well-studied and widely deployed constructions capable of supporting "partial verification" of resources as they are downloaded. Tiger tree hashes in particular are popular among several large P2P file sharing applications such as WireShare (LimeWire) [50]. Of course, a solution need not be tightly coupled to the Tiger cryptographic hashing function. The high-speed BLAKE2, SHA2, or SHA3 cryptographic hashing functions would perform just as well, if not better.

2) Replacing RIs with URNs: The goal of the Resource Identifiers (RI) is very similar to that of Uniform Resource Names (URN). It may make sense to replace the mapping between RIs and Authoritative Checksums with purely URN-based DNS lookups that return specially formatted TXT records upon success. This would further simplify the deployment process for service administrators since DNS updates would be based upon the resource's contents instead of both its contents *and where it is located physically on a distribution server*. It may also allow for additional confirmation methods of the identical resources in different domains and in different locations.

We did not choose a URN-based scheme in our initial approach due to a new URN scheme requiring the registration

Fig. 1: APNIC estimate of the percentage of global DNS resolvers (Google PDNS as well as local resolvers) performing DNSSEC validation from October 2013 to December 2018. The five year trend is positive.

of a unique identifier with the Internet Assigned Numbers Authority. Going forward, we can potentially adopt a URN scheme that already exists, such as Magnet links [35] or the informal IETF draft for hash-based URN namespaces [52]. With URNs, we can ensure our naming scheme is based solely on a resource's contents rather than both its contents *and* its arbitrary location on a web server, making DNSCHK more resilient.

VI. Conclusion

Downloading resources over the internet is indeed a risky endeavor. Resource integrity attacks, and Supply Chain Attacks more broadly, are becoming more frequent and their impact more widely felt. This paper shows that the de facto standard for addressing resource integrity risk—the use of *checksums* coupled with a secure transport layer—is an insufficient and often ineffective solution. We propose a novel resource validation scheme meant as a complete replacement for checksum based approaches: DNSCHK, which automates the tedious parts of verification to eliminate user apathy while leveraging highly-available authenticated distributed systems to ensure resources and checksums are not co-hosted. Further, we demonstrate the effectiveness and practicality of our approach versus resource integrity attacks in real-world systems.

The results of our evaluation show that DNSCHK is more effective than checksums and other existing solutions at mitigating resource integrity attacks against arbitrary resources on the internet. Further, we show DNSCHK detects a wide variety of real-world integrity errors across a diverse set of platforms. DNSCHK is both scalable and immediately deployable for organizations that secure their DNS zone(s) with DNSSEC.

We make our DNS implementation of DNSCHK available open source so that others can extend it or compare to it. Our hope is that this work motivates further exploration of resource integrity Supply Chain Attack mitigation methods.[3]

[3]The DNSCHK Chrome extension is available at https://tinyurl.com/dnschk-actual

REFERENCES

[1] D. E. E. 3rd. *Domain Name System Security Extensions*. RFC 2535. http://www.rfc-editor.org/rfc/rfc2535.txt. RFC Editor, 1999. URL: http://www.rfc-editor.org/rfc/rfc2535.txt.

[2] D. Akhawe, F. Braun, J. Weinberger, and F. Marier. *Subresource Integrity*. W3C Recommendation. http://www.w3.org/TR/2016/REC-SRI-20160623/. W3C, 2016.

[3] D. Akhawe and A. P. Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness." In: *USENIX security symposium*. Vol. 13. 2013.

[4] R. Anderson and E. Biham. "Tiger: A fast new hash function". In: *International Workshop on Fast Software Encryption*. Springer. 1996, pp. 89–97.

[5] T. Anderson. "Linux Mint hacked: Malware-infected ISOs linked from official site". In: *The Register* (2016). URL: https://www.theregister.co.uk/2016/02/21/linux_mint_hacked_malwareinfected_isos_linked_from_official_site.

[6] APNIC. *Why DNSSEC deployment remains so low*. 2017. URL: https://blog.apnic.net/2017/12/06/dnssec-deployment-remains-low/.

[7] APNIC. 2018. URL: https://stats.labs.apnic.net/dnssec/XA?c=XA&x=1&g=1&r=1&w=7&g=0.

[8] A. Barstow and E. U. *Re: [fileapi-directories-and-system/filewriter]*. 2014. URL: http://lists.w3.org/Archives/Public/public-webapps/2014AprJun/0012.html.

[9] R. J. Bayardo and J. Sorensen. "Merkle Tree Authentication of HTTP Responses". In: *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*. WWW '05. Chiba, Japan: ACM, 2005, pp. 1182–1183. ISBN: 1-59593-051-5. DOI: 10.1145/1062745.1062929. URL: http://doi.acm.org/10.1145/1062745.1062929.

[10] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. *Transport Layer Security (TLS) Extensions*. RFC 3546. RFC Editor, 2003.

[11] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *OpenPGP Message Format*. RFC 4880. http://www.rfc-editor.org/rfc/rfc4880.txt. RFC Editor, 2007. URL: http://www.rfc-editor.org/rfc/rfc4880.txt.

[12] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 20162021 White Paper*. Tech. rep. Cisco, 2017. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[13] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. http://www.rfc-editor.org/rfc/rfc5280.txt. RFC Editor, 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[14] D. Crocker, P. Hallam-Baker, and T. Hansen. *DomainKeys Identified Mail (DKIM) Service Overview*. RFC 5585. 2009. DOI: 10.17487/RFC5585. URL: https://rfc-editor.org/rfc/rfc5585.txt.

[15] *CVE-2012-5159*. Available from MITRE, CVE-ID CVE-2012-5159. 2012. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5159.

[16] J. Daemen and V. Rijmen. "AES proposal: Rijndael". In: (1999).

[17] T. Dai, H. Shulman, and M. Waidner. "Dnssec misconfigurations in popular domains". In: *International Conference on Cryptology and Network Security*. Springer. 2016, pp. 651–660.

[18] J. Damas, M. Graff, and P. Vixie. *Extension Mechanisms for DNS (EDNS(0))*. STD 75. RFC Editor, 2013.

[19] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246. http://www.rfc-editor.org/rfc/rfc2246.txt. RFC Editor, 1999. URL: http://www.rfc-editor.org/rfc/rfc2246.txt.

[20] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. http://www.rfc-editor.org/rfc/rfc5246.txt. RFC Editor, 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[21] V. Dukhovni and W. Hardaker. *The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance*. RFC 7671. RFC Editor, 2015.

[22] P. Faltstrom, R. Austein, and P. Koch. *Design Choices When Expanding the DNS*. RFC 5507. RFC Editor, 2009.

[23] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://www.rfc-editor.org/rfc/rfc7231.txt. RFC Editor, 2014. URL: http://www.rfc-editor.org/rfc/rfc7231.txt.

[24] E. Foudil and Y. Shafranovich. *A Method for Web Security Policies*. Internet-Draft draft-foudil-securitytxt-04. http://www.ietf.org/internet-drafts/draft-foudil-securitytxt-04.txt. IETF Secretariat, 2018. URL: http://www.ietf.org/internet-drafts/draft-foudil-securitytxt-04.txt.

[25] Google. *Chrome APIs*. URL: https://developer.chrome.com/apps/api_index.

[26] Google. *Google IPv6 Statistics*. URL: https://www.google.com/intl/en/ipv6/statistics.html.

[27] HandBrake. *Mirror Download Server Compromised*. 2017. URL: https://forum.handbrake.fr/viewtopic.php?f=33&t=36364.

[28] "HandBrake hacked to drop new variant of Proton malware". In: *MalwareBytes Labs* (2017). URL: https://blog.malwarebytes.com/threat-analysis/mac-threat-analysis/2017/05/handbrake-hacked-to-drop-new-variant-of-proton-malware/.

[29] "Havex". In: *New Jersey Cybersecurity and Communications Integration Cell Threat Profiles* (2017). URL:

https://www.cyber.nj.gov/threat-profiles/ics-malware-variants/havex.

[30] A. Herzberg and H. Shulman. "DNSSEC: Security and availability challenges". In: *Communications and Network Security (CNS), 2013 IEEE Conference on*. IEEE. 2013, pp. 365–366.

[31] A. Herzberg and H. Shulman. "Towards Adoption of DNSSEC: Availability and Security Challenges." In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 254.

[32] P. Hoffman and J. Schlyter. *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698. http://www.rfc-editor.org/rfc/rfc6698.txt. RFC Editor, 2012. URL: http://www.rfc-editor.org/rfc/rfc6698.txt.

[33] S. Isasi. *Expanding DNSSEC Adoption*. 2018. URL: https://blog.cloudflare.com/automatically-provision-and-maintain-dnssec/.

[34] S. Josefsson. *Storing Certificates in the Domain Name System (DNS)*. RFC 4398. RFC Editor, 2006.

[35] *MAGNET-URI Project*. URL: http://magnet-uri.sourceforge.net/.

[36] G. Markham. *Link Fingerprints*. 2008. URL: http://www.gerv.net/security/link-fingerprints/.

[37] R. C. Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[38] P. Mockapetris. *Domain names - concepts and facilities*. STD 13. http://www.rfc-editor.org/rfc/rfc1034.txt. RFC Editor, 1987. URL: http://www.rfc-editor.org/rfc/rfc1034.txt.

[39] P. Mockapetris. *Domain names - implementation and specification*. STD 13. http://www.rfc-editor.org/rfc/rfc1035.txt. RFC Editor, 1987. URL: http://www.rfc-editor.org/rfc/rfc1035.txt.

[40] J. Myers and M. Rose. *The Content-MD5 Header Field*. RFC 1864. RFC Editor, 1995.

[41] N. Nelson. "The impact of dragonfly malware on industrial control systems". In: *SANS Institute* (2016).

[42] NIST. *Cyber Supply Chain Risk Management*. URL: https://csrc.nist.gov/Projects/Supply-Chain-Risk-Management.

[43] NIST. *Estimating IPv6 and DNSSEC Deployment Status*. URL: https://usgv6-deploymon.antd.nist.gov/govmon.html.

[44] NIST. *Software Supply Chain Attacks*. 2017. URL: https://csrc.nist.gov/csrc/media/projects/supply-chain-risk-management/documents/ssca/2017-winter/ncsc_placemat.pdf.

[45] "phpMyAdmin corrupted copy on Korean mirror server". In: *SourceForge Official Blog* (2012). URL: https://sourceforge.net/blog/phpmyadmin-back-door/.

[46] "PMASA-2012-5". In: *PMASA-2012-5* (2012). URL: https://www.phpmyadmin.net/security/PMASA-2012-5/.

[47] E. Rescorla. *HTTP Over TLS*. RFC 2818. http://www.rfc-editor.org/rfc/rfc2818.txt. RFC Editor, 2000. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

[48] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. http://www.rfc-editor.org/rfc/rfc6347.txt. RFC Editor, 2012. URL: http://www.rfc-editor.org/rfc/rfc6347.txt.

[49] M. Richardson. *A Method for Storing IPsec Keying Material in DNS*. RFC 4025. RFC Editor, 2005.

[50] *ShareWire (LimeWire PE) Tiger Tree Hash Implementation*. 2010. URL: https://github.com/metapirate/LimeWire-Pirate-Edition/tree/a12327753c0a72b815b5e828dd1be9f39b2b7fc9/components/gnutella-core/src/main/java/com/limegroup/gnutella/tigertree.

[51] L. M. Team. *Beware of hacked ISOs if you downloaded Linux Mint on February 20th!* 2016. URL: https://blog.linuxmint.com/?p=2994.

[52] P. Thiemann. *A URN Namespace For Identifiers Based on Cryptographic Hashes*. Internet-Draft draft-thiemann-hash-urn-01. Archived. Internet Engineering Task Force, 2003. 10 pp. URL: https://datatracker.ietf.org/doc/html/draft-thiemann-hash-urn-01.

[53] Z. Whittaker. "Hacker explains how he put "backdoor" in hundreds of Linux Mint downloads". In: *ZDNet* (2016). URL: https://www.zdnet.com/article/hacker-hundreds-were-tricked-into-installing-linux-mint-backdoor/.

[54] A. Whitten and J. D. Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0". In: *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=1251421.1251435.

[55] P. Wilson. *How bad is IPv4 address exhaustion?* 2018. URL: https://blog.apnic.net/2018/02/15/bad-ipv4-address-exhaustion/.

[56] P. Wouters. *DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP*. RFC 7929. RFC Editor, 2016.

[57] Y. Yao, L. He, and G. Xiong. "Security and cost analyses of DNSSEC protocol". In: *International Conference on Trustworthy Computing and Services*. Springer. 2012, pp. 429–435.