HASCHK: Automating Checksum Verification And Defeating Co-Hosting Leveraging High Availability Systems

Anonymous Author(s)

Abstract

Downloading resources over the internet comes with many risks, including the chance that the resource has been corrupted, or that an attacker has replaced your desired resource with a compromised version. The de facto standard for addressing this risk is the use of *checksums* coupled with a secure transport layer; users download a resource, compute its checksum, and compare that with an authoritative checksum. Problems with this approach include (1) *user apathy*—for most users, calculating and verifying the checksum is too tedious; and (2) *co-hosting*—an attacker who compromises a resource can trivially compromise a checksum hosted on the same system. The co-hosting problem remains despite advancements in tools that automate checksum verification and generation. In this paper we propose *HASCHK*, a resource verification protocol expanding on de facto checksum-based integrity protections to defeat co-hosting while automating the tedious parts of checksum verification. We evaluate HASCHK's security, practicality, and ease of deployment by implementing a proof-of-concept Google Chrome extension. Our implementation is tested versus common resource integrity violations. We find our approach is more effective than existing mitigation methods, significantly raises the bar for the attacker, and is deployable at scale.

1 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 [1, 2]. 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 [3, 4]. 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, 6]. 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 [7]. HandBrake developers recommended users perform *checksum verification* to determine if their install was compromised [8].

Clearly, downloading resources over the internet (TCP/IP) comes with considerable risk. This risk can be divided into three broad concerns: 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 [9]. Communication confidentiality allows us to keep the data transacted between two or more parties private through some form of encryption, such as AES [10]. 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 [9, 11–14]. Assuming well behaved certificate authorities and modern browsing software, TLS and related protocols, when properly deployed, mitigate myriad attacks on authentication and confidentiality.

However, as a *communication* protocol, TLS only guarantees the integrity of each *end-to-end communication* via message authentication code (MAC) [11]. But protected encrypted communications mean nothing if the contents of those communications are corrupted before the fact. Hence, attacks against the integrity of resources at the application layer (rather than the transport layer) are outside the threat model addressed by TLS and HTTPS [11, 14].

Attacks on resource integrity can be considered a subset of *Supply Chain Attacks* (SCA). Rather than attack an entity directly, SCAs are the compromise of an entity'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 [15]. These attacks are hard to detect, even harder to prevent, and have the goal of infecting and exploiting targets and victims by abusing the trust between consumer and reputable software vendor [16].

Ensuring the integrity of resources exchanged over the internet despite SCAs and other active attacks is a hard and well studied problem [9, 14, 17–24]. The de facto standard for addressing this risk in the generic case is using *checksums* coupled with some secure transport medium like TLS/HTTPS. Checksums in this context are cryptographic digests generated by a cryptographic hashing function [25] 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 to yield a local checksum and then match it with a checksum given to them by some trusted authority.

However, checksums come up short as a solution to the resource integrity problem. Foremost is a well-understood but harsh reality: **user apathy**. Most users will not be inconvenienced with manually calculating checksums for the resources they download [23, 26]; moreover, most users will not take the time to understand how checksums and integrity verification work [23, 27, 28]. 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?" [29]. Hardly unique to checksums, designers of security schemes from HTTPS to PGP to Google Chrome dangerous download warnings have found user apathy a difficult problem space to navigate [23, 30–38].

Even if a user feels the urge to manually calculate and verify a resource's checksum, they must search for a corresponding "authoritative checksum" to compare against. 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 [23]; users are not guaranteed to find an authoritative checksum, even if they are published online somewhere, even if they appear on the same page as the resource they are trying to protect. If users do manage to find the authoritative checksum manually and also recognize the checksums are different, the user is then expected to "do the right thing," whatever that happens to be in context.

A major contributor to this confusion is the tradeoff made by **co-hosting** a resource and its authoritative checksum on the same distribution system, *e.g.* a web page hosting both a hyperlink pointing to a resource and that resource's authoritative checksum together. While cost-effective for the provider and less confusing for the user, an attacker that compromises a system hosting both a resource and its checksum together can mutate both, rendering the checksum irrelevant [24]. This is true for automated checksum verification solutions as well [23]. The co-hosting problem was demonstrated by the 2016 hack of Linux Mint's distribution server [5, 6].

For these reasons, checksums as they are typically deployed are not very effective at guaranteeing resource integrity, even if automatic verification by web clients is attempted. Recognizing this, some corporations and large entities rely instead on approaches like digital signature verification, code signing, and Binary Transparency [9, 39]. These roll-your-own solutions, often proprietary, have been deployed successfully to mitigate resource integrity attacks in mature software package ecosystems like Debian/apt and Red Hat/yum 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 mature software package ecosystems with built-in PKI support. In the United States for instance, most internet users download software directly from websites or other locations [23, 40]. Moreover, such schemes are not compatible with one another and cannot scale to secure arbitrary resources on the internet without significant cost and implementation/deployment effort.

This paper addresses these problems by proposing HASCHK, a novel protocol for verifying the integrity of arbitrary resources downloaded over the internet that is a complete replacement for typical checksum-based schemes, significantly raises the bar for the attacker, and can be implemented in more than just browser software. To overcome the challenges posed by user apathy and co-hosting, HASCHK is implemented in two parts: a backend for resource *providers* and a frontend for resource *consumers*—*i.e.* (end) users. Providers use a high availability backend to advertise which resources they provide for download. To defeat co-hosting, this backend exists separately from the system offering those resources. To account for user apathy, the frontend client automatically computes a (non-authoritative) checksum identifying the resource, queries the backend using that checksum, and mitigates the threat to the user in the case where the download is deemed compromised. Hence, HASCHK consists of both the protocol by which these pieces communicate and their implementation.

We approach these problems with four key concerns in mind. (1) HASCHK frontends must provide security guarantees transparently without adding any extra user burden in the common case. Here, an optimal implementation avoids relying on the user to overcome apathy in the interest of security while accounting for the tendency of some users to click through security warnings so as to not be inconvenienced. The former is achieved through automation and the latter through careful design. (2) HASCHK must be low effort for providers to integrate and deploy in concert with the resource(s) they are meant to protect. There must be no requirement to configure a secondary system solely dedicated to hosting checksums. (3)

The protocol is not tightly coupled with any particular high availability system. (4) Neither application, website source code changes, user-facing server, nor web infrastructure modifications are necessary.

To demonstrate the general applicability of our protocol, we implement a frontend Google Chrome extension and two different high availability backends: one based on the public Domain Name System (DNS) [41, 42] via Google DNS [43] and another based on the Ring OpenDHT network [44, 45] via a custom Representational State Transfer (REST) API. Of course, these HASCHK component implementations should be considered proof-of-concept.

We evaluate the security, performance impact, scalability, and deployment overhead of our implementation using a publicly available patched HotCRP instance. While not a panacea, we show that our implementation is capable of detecting real-world resource compromises, even when the server is under the attacker's control. Additionally, we find no practical obstacles to efficient deployment at scale or to security outside of those imposed by the Chrome V2 Extension API.

In summary, our primary contributions are:

First, we propose a practical automated approach to ensuring the integrity of arbitrary downloads. Our protocol requires no modifications to web standards/infrastructure or source code, does not employ unreliable heuristics, does not expect checksums to be co-hosted on the same page as do other automated approaches, and can be transparently implemented. Further, our protocol can be used by more than just browsers; HASCHK can protect downloads in FTP clients, wget/curl clients, and other software.

Second, we present our proof-of-concept implementation, a Google Chrome extension, and demonstrate our protocol's effectiveness empirically. We show that our implementation is capable of detecting resource compromises when the server is under the attacker's control, thus significantly raising the bar for an attacker. We additionally make our frontend implementation and OpenDHT JSON API available open source (see Section 8).

Third, we evaluate our implementation and find marginal integration and deployment overhead with no practical obstacles to scalability or security outside those imposed by the Chrome API. To the best of our knowledge, this is the first approach that is not susceptible to the pitfalls of co-hosting. Hence, we conclude that HASCHK is more effective at detecting resource integrity attacks than manual checksum verification and prior automated schemes.

2 Background

In this section, we discuss Supply Chain Attacks (SCAs), including four case studies describing real-world resource integrity attacks that take advantage of the ineffectiveness of checksums. These motivate the HASCHK approach.

2.1 Resource Integrity SCAs

Modern software requires a complex globally distributed supply chain and development ecosystem for organizations and other providers to design, develop, deploy, and maintain products efficiently [16]. 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, download mirrors, 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 [16, 24]. These types of compromises are known as Supply Chain Attacks (SCA). In the context of software development, SCAs are the compromise of an entity's software source code via cyber attack, insider threat, upstream asset/package management 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 [15].

Every year, major SCAs become more frequent and their fallout more widely felt [15, 16]. 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.

Table 1 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.

2.2 Motivation: Cases

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

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 [46]. The administrator of the mirror in question confirmed the attack was due to a vulnerability not shared by SourceForge's other mirrors [4].

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 [3]. 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 organi-

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

Table 1: The generic software engineering supply chain.

zations [4].

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 [4]. 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. 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 [5]. Attackers mutated download links originally pointing to the Linux Mint 17.3 Cinnamon edition ISO, redirecting unsuspecting users to a separate system hosting a custom Mint ISO compiled with the IRC-based Linux backdoor malware *Tsunami* [6]. The attack affected hundreds 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 [29]. The authoritative checksum displayed on the official website was also mutated to corroborate the backdoored ISO [29], 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 cannot find it or 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 [1, 2]. 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 [2]. 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, *e.g.* 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 [2]. The motivation behind the Havex malware was intelligence exfiltration and espionage [1]. How many of these vendors employed checksums and other mitigations as part of their software release cycle is not well reported, though investigators note said vendors' distribution mechanisms were insecure [2]; however, an automated resource verification method could have helped mitigate the delivery of compromised software to 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 [7]. The number of users potentially affected is unreported.

The goal of the attack was the exfiltration of victims' sensitive data, including decrypted keychains, private keys, browser password databases, 1Password/Lastpass vaults, decrypted files, and victims' personal videos and other media [7]. The HandBrake developers recommended users perform manual checksum verification to determine if their installation media was compromised [8].

Despite the attackers mutating the HandBrake binary, the authoritative checksums listed on the official HandBrake download page were reportedly left untouched [8]. Further, the developers of HandBrake store their authoritative checksums both on their official website and in their official GitHub repository [8]. 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. Which are the actual authoritative checksums? On top of requiring the technical knowledge, a user in this confusing situation is then expected to "do the right thing," whatever that happens to be in context.

3 Related Work

In this section, we examine prior approaches to guaranteeing resource integrity over the internet. We then highlight some drawbacks to these approaches and how HASCHK differs.

Anti-malware software, heuristics, and blacklists. Anti-malware software is a heuristic-based program 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 presence of the Proton malware [7]. In the 2012 compromise of SourceForge's CDN, the malicious changes to the phpmyAdmin image do not appear as malware to anti-malware software [3].

Similarly, all modern browsers employ heuristic and blacklist-based detection and prevention schemes in an attempt to protect users from malicious content on the internet. The warnings generated by browser-based heuristics and blacklists are also reactive rather than proactive; hence, they are generally ineffective at detecting active or novel attacks on the integrity of the resources downloaded over the internet.

On the other hand, HASCHK relies on no heuristics or blacklists and is not anti-malware software. HASCHK is a protocol for automating checksum verification of resources. This insures download integrity—that a user is receiving the expected resource a provider is advertising—not that the expected resource is not malware.

Link Fingerprints and Subresource Integrity. The Link Fingerprints (LF) draft describes an early HTML hyperlinks and URI based resource integrity verification scheme that "provides a backward-compatible technique for resource providers to ensure that the resource originally referenced is the same as the resource retrieved by an end user." [20]. The World Wide Web Consortium's (W3C) Subresource Integrity (SRI) describes a similar HTML-based scheme designed exclusively with CDNs and web assets in mind.

Like HASCHK, both LF and SRI employ cryptographic digests to ensure no changes of any kind have been made to a resource [19]. Unlike HASCHK, LF and SRI apply *only to resources referenced by script and link HTML elements*; HASCHK, on the other hand, can ensure the integrity of *any arbitrary resource downloaded over the internet*, even outside of HTML web pages and browser software. Further, the checksums contained in the HTML source must be accurate for SRI to work. If the system *behind* the CDN is compromised, the attacker can alter the HTML and inject a malicious checksum or even strip checksums from the HTML entirely. With HASCHK, however, an attacker would additionally have to compromise the separate backend system that advertises the provider's resources, thus raising the bar.

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 [18]. Further, the header could be easily stripped off or modified by proxies and other intermediaries [17]. HASCHK exhibits none of these weaknesses.

Deterministic Build Systems and Binary Transparency. A deterministic build system is one that, when given the same source, will deterministically output the same binary on every run. For example, many packages in Debian [47] and Arch Linux can be rebuilt from source to yield an identical byte for byte result [48], allowing for verification of the *Integration* and perhaps *Development* supply chain phases (see Table 1). Further, using a merkle tree [49] or similar construction, an additional chain of trust can be established that allows for public verification of the *Deployment*, *Maintenance*, and *Retirement* supply chain phases. Companies such as Mozilla refer to the latter as "Binary Transparency."

Like HASCHK, binary transparency establishes a public verification scheme that allows third party consumers access to a listing of source updates advertised by a provider [39]. Consumers can leverage deterministic build systems and Binary Transparency together to ensure their software is the same software deployed to every other system. Unlike HASCHK, binary transparency only allows a user to verify the integrity of *source updates to binaries*; our protocol allows a user to verify the integrity of *any arbitrary resource* while specifically addressing co-hosting.

Stickler and Cherubini et al. Stickler [24] by Levy et al. is an automated JavaScript-based stand-in for SRI for protecting the integrity of web application files hosted on CDNs. Stickler does not require any modifications to the client (*i.e.* a frontend), instead delivering a bootloader to load and verify resources signed before the fact. However, as it was designed to stand-in for SRI, Stickler inherits some of SRI's limitations. Specifically: Stickler was not designed to protect arbitrary resource downloads and, if the publisher's server is compromised, Stickler's bootloader can be stripped out of the initial HTTP response altogether. Something like this is not possible with HASCHK.

The automated checksum verification approach described by Cherubini et al. [23], also based on SRI, is similarly vulnerable to (and relies upon) co-hosting. Cherubini's browser extension works by both looking for embedded checksums in download links (SRI) and extracting hexadecimal strings that look like checksums directly from the HTML source. An attacker, after compromising the resource file, need only modify the provider's HTML file to inject a corrupted "integrity" attribute containing a checksum matching that corrupted resource, causing Cherubini's extension to misreport the dangerous download as safe [23]. Additionally, Cherubini's extension: (1) does not alert users when corresponding authoritative checksums are not found, which means an attacker can simply strip all checksums from the server response to pass off compromised resources to users; (2) considers a download "safe" so long as *any checksum found on the page matches it*, which means an attacker can just inject the compromised checksum somewhere in the HTML source alongside the legitimate checksum to similarly pass off compromised resources to users; (3) does not support direct downloads, *i.e.* when a user enters a resource's URI into the browser manually rather than click a hyperlink. None of this is a problem for HASCHK.

4 Security Goals and Threat Model

The goal of HASCHK is to ensure the integrity of arbitrary resources downloaded over the internet, even in the case where the system hosting those resources is completely compromised. Given a HASCHK-aware client and a resource provider with a conforming backend, the user must be automatically warned whenever what they are downloading does not match a corresponding authoritative checksum. Given proper deployment, HASCHK should, to whatever extent possible, operate without issuing false negatives, *i.e.* compromised resources that do not trigger a warning, or false positives, *i.e.* benign resources that do trigger a warning.

We consider the following entities: (1) a HASCHK-aware client as the *frontend*, (2) a *server* (or servers) as the distribution system hosting the *resource*, and (3) a separate high availability system implementing HASCHK as the *backend*. Specifically, we consider a *generic frontend* that might be any web-facing software, such as FileZilla, and a *browser frontend* such as Google Chrome or Mozilla Firefox.

For a generic frontend, we assume the adversary can make the server respond to a resource request with any resource, including a compromised version of a resource. We also assume the adversary can tamper with any other server response, including adding, removing, or manipulating one or more checksums if they appear.

For a browser frontend and web server, we contemplate two scenarios. From either an existing tab/window or a new one, the user (1) directly enters the resource URI into the browser, initiating a direct download or (2) navigates to an HTML file (*web page*) on the server containing a hyperlink pointing to the resource. The user clicks this hyperlink to download the resource. This resource is hosted on the same server as the web page or externally on a third party server such as a CDN. Both are valid deployments in the context of HASCHK. We assume the adversary can manipulate the resource and web page wherever they are stored. The adversary may: (1) compromise the resource, (2) modify the hyperlink to point to a compromised resource anywhere on the internet, and/or (3) add, remove, or manipulate any checksums that appear on the web page or elsewhere (excluding the backend).

Hence, we do not trust the integrity of the server's responses in any scenario. We do, however, trust the integrity (and authenticity) of the backend's responses. Further, we expect the backend to be highly available. In Section 6, we go over the implications of a compromised backend.

As an aside, we consider here the standard non-HASCHK case where the resource is co-hosted alongside a web page containing both a hyperlink pointing to the resource *and an authoritative checksum corresponding to the resource*. As demonstrated throughout the paper, when a server co-hosting resources and checksums is compromised, it is trivial to manipulate both checksum and resource, rendering all forms of checksum verification irrelevant. Assuming proper deployment, it is the goal of HASCHK to ensure this is never the case.

5 The HASCHK Protocol

In this section, we describe the components of the HASCHK protocol, step through each in detail, and then introduce our proof-of-concept implementations of these components.

5.1 Participants

Provider. The *provider* is the entity in control of both the server and backend with the goal of ensuring the integrity of resources downloaded from their system.

HASCHK Frontend. The *frontend* is responsible for calculating the Uniform Resource Name (URN) and Backend Domain (BD) of the resource downloaded from the server. Afterwards, the frontend queries the backend using this URN and, if an integrity violation is detected, quarantines the resource and warns the user.

Server. The *server* is a distribution system (hopefully) controlled by the provider that hosts resources for download. It can be internal or external, a single server or many, first-party or third-party.

HASCHK Backend. The *backend* is responsible for advertising a queryable listing of URNs that HASCHK frontends can use to judge the legitimacy of downloads. It is a high availability system that is wholly separate from the server (not co-hosted) and controlled by the provider.

5.2 Protocol Overview

[TODO: Add Visio fig (implementation) 1 here.]

Fig. **??** gives a high level overview of the HASCHK protocol. Initially, when a user finishes downloading a resource from the provider's server, the frontend runs the resource's contents through a SHA-256 hashing function, yielding a 256-bit digest. Any hashing function can be used so long as it is pre-image and collision resistant [25]. This digest is used to construct a hash-based Uniform Resource Name (URN) uniquely identifying the resource. Additionally, a Backend Domain (BD) is derived from the domain name of the server. The frontend uses the BD in an implementation-dependent manner to locate and query the backend, checking for the existence of the constructed URN and essentially asking: *"is this a compromised resource?"* If the URN is found, a negative response is returned indicating a verified resource. If the URN is not found, a positive response is returned indicating a compromised resource.

If the query returns a positive response: (1) the user should be warned pursuant to the recommendations of Akhawe et al [31] (avoid warning fatigue, be invisible), Cherubini et al [23] (simple non-technical language, secure default action), Modic et al [35] (issue warnings from a position of authority), and others; (2) the download should be deleted, renamed, or otherwise made inaccessible/quarantined by default; and (3) the user should still be allowed to "click through" and override the warning and the default quarantine behavior, though ideally this should be less convenient than the default action [23].

If the query returns a negative response, the frontend should indicate this as inconspicuously as they can manage to mitigate the threat of warning/popup fatigue [23, 31] and habituation [50]. For example, our frontend implementation simply changes its icon when downloads are successfully verified and only issues popup warnings when compromised downloads are positively identified.

To prevent false positives, integrity checking is skipped if the backend's location is unresolvable or HASCHK is not properly deployed. Determining when this is the case is implementation-dependent. To prevent false-negatives, when HASCHK is properly deployed and a URN is not found during lookup, the backend response will always be interpreted as positive. As a result, it is not possible to only secure "some" resources on a server.

5.2.1 Deriving the Backend Domain (BD)

Before we can communicate with a provider's backend, we require some scheme to locate it. We refer to this scheme as deriving the *Backend Domain* (BD). The BD is some identifier that allows the frontend to locate the backend. For our Domain Name System (DNS) based implementation (below), the BD is the Third-Level Subdomain (3LD) or Second-Level Domain (2LD) based on the server URI's host subcomponent [51] and are queried in that order.

Example 1: an FTP frontend (*e.g.* Filezilla, Transmit, et cetera) downloading a resource from an FTP server at `ftps://un:pw@ftp.example1.com:8080/some/resource` has a host subcomponent of `ftp.example1.com` and a BD of `ftp.example1.com` or `example1.com`.

Example 2: a browser frontend (*e.g.* Lynx, Google Chrome, Internet Explorer, et cetera) downloading a resource at `https://s4.l.cdn.example2.com/fl` from a hyperlink on the page `https://dl.app.example3.com/index/page` has a host subcomponent of `dl.app.example3.com` (*not* `s4.l1.cdn.example2.com`) and a BD of `app.example3.com` or `example3.com`.

It is important that the right BD is derived to ensure correct operation of the protocol. This concern is implementation-dependent. Specifically, in the case of the browser frontend in example 2 (above), the BD is not derived from the resource's URI directly but from the URI of the HTML page containing the hyperlink pointing to that resource. This prevents an adversary from trivially fooling HASCHK by, for instance, replacing `https://s4.l.cdn.example2.com` with a hyperlink pointing to `https://attacker.com` and hosting a conforming HASCHK backend that would cause the frontend to respond with a false negative.

Further, depending on the implementation, a frontend or backend may not be using URIs and DNS to transact information over the internet at all. An example of this is a purely Distributed Hash Table (DHT) based backend. In such a case, any derivation algorithm (or no derivation at all, *e.g.* the BD is hardcoded) can be used as long as correct operation of the protocol is ensured.

5.2.2 Constructing Uniform Resource Names (URN)

To ensure the integrity of arbitrary resources, we require some method to uniquely and durably identify those resources. We accomplish this through the adoption of the informal IETF draft for the construction of hash-based *Uniform Resource Name* (URN) namespace [52], which the frontend follows when calculating URNs for each resource downloaded. Through whatever implementation-specific method, the backend must "advertise" or allow lookups against a set of expected URNs corresponding to the resources hosted on the provider's server, even if those resources have unstable access paths or URIs; *e.g.*, when resources are hosted externally, on download mirrors, on CDNs, et cetera. This requirement is satisfied by implementations combining URNs with the BD, thus durably associating the URNs calculated in the frontend with the URNs advertised by a provider's backend regardless of where the corresponding resources are hosted on the provider's server.

Since the backend advertising URNs is *never* co-hosted alongside the system that distributes the corresponding resources—like a web or FTP server—HASCHK retains the ability to protect users from dangerous downloads *even when the system distributing the resource has been completely compromised*. This is not true of prior approaches to automated checksum verification of arbitrary resources [23].

5.3 Protocol Implementation

[TODO: Add Visio fig 2 (implementation) here.]

We implement a proof-of-concept HASCHK frontend as a Google Chrome extension. To demonstrate the general applicability of our approach, our frontend currently works with two backends: (1) directly with DNS via Google's JSON API for DNS over HTTPS (DoH) and (2) indirectly with DHT (Ring OpenDHT) via a local Representational State Transfer (REST) API we designed to mimic the response syntax of Google's DoH JSON API. Other candidate backends include storage clusters, relational and non-relational databases, and any high availability key-value store.

Fig. **??** gives a high level overview of how our implementation works. First, the user initiates a download either directly (*e.g.* typing a URI manually) or by following a hyperlink. Our extension, having observed the web request earlier, associates that request with the new download item. The original request may have directly triggered the download or the download may have been triggered after one or more redirections. Our extension derives the correct BD in both cases (see below). At this point, if the backend does not exist or does not conform to the protocol, the protocol terminates. Otherwise, after the download completes, a URN is calculated and combined with the BD to make a final query to the backend. In effect, this query asks: *is this resource compromised?* If the response to the query is negative (*i.e.* a TXT record (DNS) or data value (DHT)), the extension considers the resource validated. On the other hand, if the response to the query is positive (*i.e.* no record found or bad data value returned), the extension considers the resource compromised, deletes the resource file, and warns the user.

In keeping with UX suggestions of prior work, our extension is virtually transparent to end users in the common case that a download is successfully verified by a provider's backend or when HASCHK has not been properly deployed by a provider. However, in the relatively uncommon case that a resource is deemed compromised, the extension will delete the file and alert the user with the primary option to "dismiss" the alert, mimicking Chrome's own highly effective dangerous download warning [32]. Specifically: the user has the option of clicking through the warning via a low-key secondary interface where they can force the extension to ignore the compromised nature of the resource *the next time it is downloaded*, forcing the user to trigger the download again. This inconvenience, favoring users' security over choice, is in keeping with the recommendations of prior work regarding security warning UX (see Section 1).

As no JavaScript OpenDHT client implementation exists, we developed a REST API to facilitate communication between our extension and the Ring OpenDHT network via HTTP and JSON. In a production implementation, a JavaScript OpenDHT client would be baked directly into the extension.

Deploying the DNS backend is as simple as adding certain TXT records to our DNS zone. This is a straightforward operation. Similarly, deploying the DHT backend requires adding certain key-value to the OpenDHT network, which is trivial. Moreover, both DNS and DHT backends are performant and highly available. The Ring OpenDHT network is also free to use. In the case of DNS, the vast majority of web-facing providers and IT teams are already using it, already pay for their own DNS zones, and are already quite familiar with configuring and managing them. Hence, no exotic secondary backend system is required for providers with web servers. Additionally, we argue updating the URNs stored by these backends automatically—by integrating a URN calculation and record insertion/update step into a modern development toolchain or resource deployment pipeline—is relatively low-effort and straightforward for providers.

No application or website source code changes, costly user-facing server or web infrastructure customizations, or modifications to web standards are needed to enable our extension to protect a provider's resources. Given a production-ready implementation of our frontend—coupled with the near ubiquitous adoption of DNS—HASCHK (with a DNS backend) could be deployed immediately by interested providers.

5.3.1 URNs, BDs, and Fallthrough in Google Chrome

BD derivation in the browser is non-trivial. Users can initiate downloads through clicking links inside *and outside* the browser (*e.g.* in an email application), through asynchronous JavaScript, and by entering URIs into the browser manually. To catch these and other edge cases, we implement our extension using the WebRequest API [53], which allows us to monitor all navigation events, collate redirects into an interrogable chain of requests, and associate new download items with the URI where they were originally initiated.

To query the backend, we: (1) BASE32 encode the resource's URN. (2) Divide the 112-character result into two 56-character strings `C1` and `C2`. (3) We calculate the BD, which is either the 3LD and 2LD from `details.originUrl`. We choose by issuing up to two queries of the form `AL.BD`, where the Application Label (AL) is the constant string `_haschk`, looking for a response containing the value `OK`. We first query the 3LD as BD and, if we do not receive the proper response there, query the 2LD as BD. If the proper response is not received from either query, the extension assumes HASCHK has not been properly deployed and terminates. (4) Otherwise, concatenating C1, C2, AL, and BD, we issue a query of the form `C1.C2.AL.BD`, again looking for a response containing `OK` (negative) or that the query was not found (positive).

To handle redirection, we use Chrome's Web Request API to associate redirected requests with one another. The originating request's `origin` at the top of this chain is used to derive the BD using the `details.originUrl` and `details.documentUrl` properties (also avoiding iframe issues). This prevents an adversary from trivially fooling HASCHK by replacing a legitimate hyperlink with a malicious one. A naive implementation might use the `DownloadItem.referrer` property to derive the BD, since it should be populated with the originating URL. Unfortunately, this property can be manipulated with one or more redirections causing the BD to point into the attacker's DNS zone. If the attacker properly deploys a HASCHK backend, they could then trivially force false negatives.

Keeping a chain of associated requests also allows us to implement "fallthrough" functionality where, if the original request at the head of the chain has not deployed HASCHK, the extension can walk the chain, checking each point of redirection for a proper HASCHK deployment. This would ensure URL shorteners and other redirection services do not trigger false positives.

6 Evaluation

In this section, we evaluate our implementation empirically using a patched HotCRP instance and random sampling of papers published in previous USENIX proceedings. We then examine the performance impact, deployment overhead, and scalability of our implementation. We show that HASCHK is more effective than existing approaches at detecting integrity problems in arbitrary downloads over the internet.

6.1 Security and Performance

To empirically evaluate our implementation, we launch a lightly modified version of the popular open source research submission and peer review software, HotCRP (version 2.102; see Section 8). Our modifications allow anyone visiting the site to interactively corrupt submissions and manipulate relevant DNS entries at will.

For our evaluation, we upload 10 different USENIX PDFs to our HotCRP instance. Upon their upload, HotCRP calculated and displayed the unique checksum (a SHA-256 digest) of each PDF. After each PDF is uploaded, we immediately download it and manually calculate a checksum locally, matching each to the checksum displayed by the HotCRP software. Next, we utilize the custom functionality we patched into HotCRP to populate our DNS backend with each file's expected URN.

After installing the frontend into our Google Chrome browser, we again download each file. For each observed download, our extension indicated a "safe" judgement. We then utilize our patched functionality to add junk data onto the end of each of the uploaded PDFs, thus corrupting them. We also modified HotCRP so that it updated the displayed checksums to match their now-corrupted counterparts.

Once again, we download each file. Our extension reported an "unsafe" judgement (a true positive) for each corrupted PDF file. Calculating the local checksum and checking it against the value reported by our HotCRP instance leads to a match (a false negative; *i.e.*, the result of co-hosting), defeating prior approaches to both manual and automated checksum verification.

We ran this experiment three times, each with different sets of PDFs and using both the DNS and DHT based backends. We observed consistent results.

Finally, we utilize our patched functionality to test a "redirection" attack where the hyperlink pointing to the correct PDF is replaced with a hyperlink to a dummy malware file hosted in a distinct malicious DNS zone with conforming HASCHK deployment. When an unsuspecting user clicks such a link, they expect to download the PDF document from HotCRP but are instead navigated to a PHP script that redirects the request one or more times before landing on the dummy malware file. This process corrupts Chrome's `DownloadItem.referrer` property. Even in this case, our implementation correctly flagged this download as suspicious once the download began, successfully warning the user.

While evaluating our implementation, we observe no additional network load or CPU usage with the extension loaded into Chrome. Measurements were taken using the Chrome developer tools. Intuitively, this makes sense given the lightweight nature of the cryptographic operations involved.

We also consider the implications of a compromised backend, such as if an attacker tampers with the provider's DNS server or its response. Since we trust the integrity and authenticity of responses from the backend, this is ultimately outside our threat model. However, in the case of a compromised DNS zone by itself, at worst the attacker can perpetuate a denial of service attack by triggering false positives. Without compromising both the backend and the distribution server, the attacker still cannot deliver compromised resources to users. Further, we argue a provider has much bigger problems if their DNS zone or other backend is compromised.

6.2 Scalability and Deployment

As HASCHK assumes a high availability backend, we conclude that the scalability of HASCHK can be reduced to the scalability of its backend. We are aware of no other obstacles to scalability beyond those inherited from the underlying backend system.

Obviously, our DNS backend relies on DNS [41]. DNS was not originally designed to transport or store relatively large amounts of data, but the content of our DNS TXT records are very small (usually two bytes or less). Further, the URNs queryable via DNS request are exactly 112 characters, *i.e.*the length of a BASE32 encoded URN, and are divided into two 53 character labels, conforming to DNS label length limits [42].

We are also unaware of any practical limitation on the number of resource records a DNS zone file can support. A

DNS server can host tens of thousands of resource records in their backend file [41, 42]. Moreover, several working groups have considered using DNS as a storage medium for checksums/hash output, so the concept is not novel. Examples include securitytxt [54] and DKIM [55].

Additionally, using DNS as a backend 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.

With the HotCRP demo, our entire resource deployment scheme consisted of (1) the addition of a new TXT entry to our DNS backend and (2) a new value published to our DHT backend during the paper (resource) submission process. We argue updating DNS record (or DHT value) during the resource deployment process is simple enough for developers and providers to implement and presents no significant burden to deployment. For reference, we implemented the functionality that automatically adds (and updates) the DNS TXT records advertising the URNs of papers uploaded to our HotCRP instance in under 10 lines of JavaScript.

6.3 Limitations

While still effective, our extension would be even more effective if Chrome/Chromium or the more general WebExtensions API allowed for an explicit `onComplete` event hook in the downloads API. This hook would fire immediately before a file download completes and the file becomes executable, *i.e.* has its `.crdownload` or `.download` extension removed. The hook would consume a `Promise`/`AsyncFunction` that kept the download in its fully-downloaded but non-complete state until said `Promise` completed. This would allow an extension to alter a download's `DangerType` property after some computation, prompting Chrome to handle alerting the user using its Dangerous Download UX [32]. This would have the advantage of communicating intent through the browser's familiar and authoritative UI and prevent the corrupted 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 our purposes because it can only be called after the download completes while leaving the file in an accessible and executable state until the method is called.

Redirection services, like URI shortening apps, might still lead to false positives even with the fallthrough functionality of a domain on the request chain has deployed HASCHK. Further, our extension does not work for PDFs and other downloads handled directly by the browser. And, given the topology of the webrequest API, iframes and similar elements may require special consideration beyond examining `details.documentUrl`.

Additionally, not all servers/backends on the internet use DNS, and our extension does not support downloads made that bypass DNS. In future work, our DHT backend would be able to handle such downloads. Further, our extension keeps 1000 requests in memory so that they can be mapped to download items later in time. This might be vulnerable to attacks involving excessive redirection and overflow.

7 Conclusion

Downloading resources over the internet is indeed a risky endeavor. Resource integrity and other Supply Chain Attacks are becoming more frequent and their impact more widely felt. In this work, we showed that the de facto standard for protecting the integrity of arbitrary resources on the internet—the use of *checksums*—is insufficient and often ineffective. We presented HASCHK, a practical resource verification protocol that automates the tedious parts of checksum verification while leveraging pre-existing high availability systems to ensure resources and their checksums are not vulnerable to co-hosting. Further, we demonstrated the effectiveness and practicality of our approach versus real-world resource integrity attacks in a production application.

The results of our evaluation show that our approach is more effective than checksums and prior work mitigating integrity attacks for arbitrary resources on the internet. Further, we show HASCHK is capable of guarding against a variety of attacks, is deployable at scale for providers that already maintain a DNS presence, and can be deployed without fear of adversely affecting the user experience of clients that are not HASCHK-aware.

Though not a panacea, we believe our protocol significantly raises the bar for the attacker. We intend to continue developing our extension and we make it available to a wide audience (see Section 8).

8 Availability

We make our HASCHK frontend and DHT backend implementations available open source for the benefit of the community[1]. We also make publicly available our evaluation environment: a patched HotCRP instance[2]. Our hope is that this work motivates further exploration of resource integrity and other SCA mitigation strategies.

[1]HASCHK component implementations: `https://haschk.dev`

[2]HASCHK's HotCRP test environment: `https://hotcrp.haschk.dev`

References

[1] “Havex”. In: *New Jersey Cybersecurity and Communications Integration Cell Threat Profiles* (2017). URL: https://www.cyber.nj.gov/threat-profiles/ics-malware-variants/havex.

[2] Nell Nelson. “The impact of dragonfly malware on industrial control systems”. In: *SANS Institute* (2016).

[3] “PMASA-2012-5”. In: *PMASA-2012-5* (2012). URL: https://www.phpmyadmin.net/security/PMASA-2012-5/.

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

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

[6] Tim 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.

[7] “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/.

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

[9] 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.

[10] Joan Daemen and Vincent Rijmen. “AES proposal: Rijndael”. In: (1999).

[11] Tim 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.

[12] Tim Dierks and Christopher 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.

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

[14] 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.

[15] 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.

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

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

[18] 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.

[19] Devdatta Akhawe, Frederik Braun, Joel Weinberger, and Francois Marier. *Subresource Integrity*. W3C Recommendation. http://www.w3.org/TR/2016/REC-SRI-20160623/. W3C, 2016.

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

[21] 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.

[22] Donald E. Eastlake 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.

[23] Mauro Cherubini, Alexandre Meylan, Bertil Chapuis, Mathias Humbert, Igor Bilogrevic, and Kévin Huguenin. “Towards Usable Checksums: Automating the Integrity Verification of Web Downloads for the Masses”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, 1256–1271. ISBN: 9781450356930. DOI: 10.1145/3243734.3243746. URL: https://doi.org/10.1145/3243734.3243746.

[24] A. Levy, H. Corrigan-Gibbs, and D. Boneh. “Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser”. In: *IEEE Security Privacy* 14.2 (2016), pp. 22–28. ISSN: 1558-4046. DOI: 10.1109/MSP.2016.32.

[25] Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *Fast Software Encryption*. Ed. by Bimal Roy and Willi Meier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388. ISBN: 978-3-540-25937-4.

[26] Michael Fagan and Mohammad Maifi Hasan Khan. "Why Do They Do What They Do?: A Study of What Motivates Users to (Not) Follow Computer Security Advice". In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. Denver, CO: USENIX Association, June 2016, pp. 59–75. ISBN: 978-1-931971-31-7. URL: https://www.usenix.org/conference/soups2016/technical-sessions/presentation/fagan.

[27] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. "Can Unicorns Help Users Compare Crypto Key Fingerprints?" In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, 3787–3798. ISBN: 9781450346559. DOI: 10.1145/3025453.3025733. URL: https://doi.org/10.1145/3025453.3025733.

[28] Hsu-Chun Hsiao, Yue-Hsun Lin, Ahren Studer, Cassandra Studer, King-Hang Wang, Hiroaki Kikuchi, Adrian Perrig, Hung-Min Sun, and Bo-Yin Yang. "A Study of User-Friendly Hash Comparison Schemes". In: *Proceedings of the 2009 Annual Computer Security Applications Conference*. ACSAC '09. USA: IEEE Computer Society, 2009, 105–114. ISBN: 9780769539195. DOI: 10.1109/ACSAC.2009.20. URL: https://doi.org/10.1109/ACSAC.2009.20.

[29] Zack 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/.

[30] Alma 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.

[31] Devdatta Akhawe and Adrienne Porter Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness". In: *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, 257–272. ISBN: 9781931971034.

[32] *Improving Chrome's Security Warnings [public]*. URL: https://docs.google.com/presentation/d/16ygiQS0_5b9A4NwHxpcd6sW3b_Up81_qXU-XY86JHc4/htmlpresent.

[33] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. "You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '08. Florence, Italy: Association for Computing Machinery, 2008, 1065–1074. ISBN: 9781605580111. DOI: 10.1145/1357054.1357219. URL: https://doi.org/10.1145/1357054.1357219.

[34] Jan Vahrenhold. "On the Importance of Being Earnest: Challenges in Computer Science Education". In: *Proceedings of the 7th Workshop in Primary and Secondary Computing Education*. WiPSCE '12. Hamburg, Germany: Association for Computing Machinery, 2012, 3–4. ISBN: 9781450317870. DOI: 10.1145/2481449.2481452. URL: https://doi.org/10.1145/2481449.2481452.

[35] David Modic and Ross Anderson. "Reading this may harm your computer: The psychology of malware warnings". In: *Computers in Human Behavior* 41 (2014), 71–79. DOI: 10.1016/j.chb.2014.09.014.

[36] Robert W. Reeder, Adrienne Porter Felt, Sunny Consolvo, Nathan Malkin, Christopher Thompson, and Serge Egelman. "An Experience Sampling Study of User Reactions to Browser Warnings in the Field". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018. ISBN: 9781450356206. DOI: 10.1145/3173574.3174086.

[37] Mario Silic and Andrea Back. "Deterrent Effects of Warnings on User's Behavior in Preventing Malicious Software Use". In: *HICSS*. 2017.

[38] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. "What the App is That? Deception and Countermeasures in the Android User Interface". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 931–948. DOI: 10.1109/SP.2015.62.

[39] *Security/Binary Transparency*. URL: https://wiki.mozilla.org/Security/Binary_Transparency.

[40] Camille Ryan. *Computer and Internet Use in the United States: 2016*. Technical Report ACS-39.

[41] 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.

[42] 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.

[43] *JSON API for DNS over HTTPS (DoH)*. URL: https://developers.google.com/speed/public-dns/docs/doh/json.

[44] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. "OpenDHT: A public DHT service and its uses". English (US). In: *Computer Communication Review* 35.4 (Oct. 2005), pp. 73–84. ISSN: 0146-4833. DOI: 10.1145/1090191.1080102.

[45] *GitHub: savoirfairelinux/opendht*. URL: https://github.com/savoirfairelinux/opendht.

[46] *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.

[47] *Reproducible Builds (Debian Wiki)*. URL: https://wiki.debian.org/ReproducibleBuilds.

[48] *Reproducible Builds: A Set Of Software Development Practices That Create An Independently Verifiable Path From Source To Binary Code*. URL: https://reproducible-builds.org/.

[49] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*. CRYPTO '87. Berlin, Heidelberg: Springer-Verlag, 1987, 369–378. ISBN: 3540187960.

[50] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. "Crying Wolf: An Empirical Study of SSL Warning Effectiveness". In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM'09. Montreal, Canada: USENIX Association, 2009, 399–416.

[51] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[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] Google. *Chrome Extension APIs*. URL: https://developer.chrome.com/extensions/devguide.

[54] Edwin Foudil and Yakov 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.

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