Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH

Karthikeyan Bhargavan
INRIA
karthikeyan.bhargavan@inria.fr

Gaëtan Leurent
INRIA
gaetan.leurent@normalesup.org

***Abstract*—In response to high-profile attacks that exploit hash function collisions, software vendors have started to phase out the use of MD5 and SHA-1 in third-party digital signature applications such as X.509 certificates. However, weak hash constructions continue to be used in various cryptographic constructions within mainstream protocols such as TLS, IKE, and SSH, because practitioners argue that their use in these protocols relies only on second preimage resistance, and hence is unaffected by collisions. This paper systematically investigates and debunks this argument.**

We identify a new class of *transcript collision* attacks on key exchange protocols that rely on efficient collision-finding algorithms on the underlying hash constructions. We implement and demonstrate concrete credential-forwarding attacks on TLS 1.2 client authentication, TLS 1.3 server authentication, and TLS channel bindings. We describe almost-practical impersonation and downgrade attacks in TLS 1.1, IKEv2 and SSH-2. As far as we know, these are the first collision-based attacks on the cryptographic constructions used in these popular protocols.

Our practical attacks on TLS were responsibly disclosed (under the name SLOTH) and have resulted in security updates to several TLS libraries. Our analysis demonstrates the urgent need for disabling all uses of weak hash functions in mainstream protocols, and our recommendations have been incorporated in the upcoming Token Binding and TLS 1.3 protocols.

I. Introduction

Hash functions, such as MD5 and SHA-1, are widely used to build authentication and integrity mechanisms in cryptographic protocols. They are used within public-key certificates, digital signatures, message authentication codes (MAC), and key derivation functions (KDF). However, recent practical attacks on MD5 and almost-practical attacks on SHA-1 have led researchers and practitioners to question whether these uses of hash functions in popular protocols are still secure.

The first collision on MD5 was demonstrated in 2005 [38], and since then, collision-finding algorithms have gotten much better. Simple MD5 collisions can now be found in seconds on a standard desktop. In response, protocol experts reviewed the use of MD5 in Internet protocols such as Transport Layer Security (TLS) and IPsec [16], [15], [3]. Despite some disagreement on the long-term impact of collisions, they concluded that most uses of hash functions in these protocols were not affected by collisions. Consequently, MD5 continues to be supported (alongside newer, stronger hash algorithms) in protocols like TLS and IPsec.

In 2009, an MD5 collision was used to create a rogue CA certificate [36], hence breaking the security of certificate-based authentication in many protocols. A variant of this attack was used by the Flame malware to disguise itself as a valid Windows Update security patch [34]. Due to these high-profile attacks, there is now consensus among certification authorities and software vendors to stop issuing and accepting new MD5 certificates. Learning from the MD5 experience, software vendors are also pro-actively phasing out SHA-1 certificates, since collisions on SHA-1 are believed to be almost practical [35].

This leaves open the question of what to do about other uses of MD5 and SHA-1 in popular cryptographic protocols. Practitioners commonly believe that collisions only affect non-repudiable signatures (like certificates), but that signatures and MACs used within protocols are safe as long as they include unpredictable contents, such as nonces [16], [15].In these cases, protocol folklore says that a second preimage attack would be required to break these protocols, and such attacks are still considered hard, even for MD5.

Conversely, theoretical cryptographers routinely assume collision-resistance in proofs of security for these protocols. For example, various recent proofs of

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23418

TLS [17], [22], [11] assume collision-resistance even though the most popular hash functions used in TLS are MD5 and SHA-1. Whom shall we believe? Either it is the case that cryptographic proofs of these protocols are based on too-strong (i.e. false) assumptions that should be weakened, or that practitioners are wrong and collision resistance is required for protocol security.

This paper seeks to clarify this situation by systematically investigating the use of hash functions in the key exchanges underlying various versions of TLS, IPsec, and SSH. We demonstrate that, contrary to common belief, collisions *can* be used to break fundamental security guarantees of these protocols. We describe a generic class of attacks called *transcript collision* attacks, and detail concrete instances of these attacks against real-world applications. In particular, we demonstrate how a man-in-the-middle attacker can impersonate TLS 1.2 clients, TLS 1.3 servers, and IKEv2 initiators. We also show how a network attacker can downgrade TLS 1.1 and SSH-2 [39] connections to use weak ciphers. We implement proofs-of-concept exploit demos for three of these attacks to demonstrate their practicality, and provide attack complexities for the others. We believe that ours are the first hash collision-based attacks on the cryptographic constructions within these protocols.

We do not claim to have found all *transcript collision* attacks in these protocols; nor do we think that our attack implementations are the most efficient. Still, our results already provide enough evidence for us to strongly recommend that weak hash functions like MD5 and SHA-1 should be immediately disabled from Internet protocols. Partly due to recommendations by us and other researchers, these hash functions and other weak constructions based on them have been removed from the draft version of the TLS 1.3 protocol.

Outline Section II introduces transcript collision attacks on authenticated key exchange protocols. Section III outlines the state-of-the-art in collision-finding algorithms for MD5, SHA-1, and their concatenation. Section IV describe the TLS protocol, and Section V describes concrete attacks on various versions of TLS and three proof-of-concept demos. Section VI describes concrete attacks on IKE and SSH. Section VII summarizes the impact of our attacks and disclosure status. Section VIII concludes.

II. Transcript Collision Attacks on Authenticated Key Exchange

Authenticated Key Exchange (AKE) protocols are executed between two parties, usually called client and server or initiator and responder, in order to establish a shared session key that can be used to encrypt subsequent messages. A typical example is the SIGMA' protocol depicted in Figure 1. This protocol is a variant

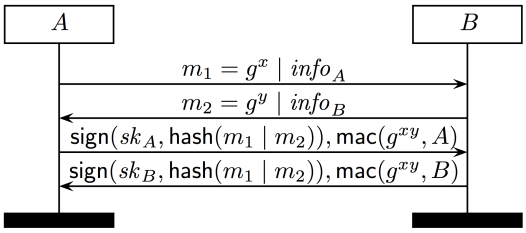

Fig. 1. SIGMA': A mutually-authenticated key exchange protocol

of the basic SIGMA (sign-and-mac) protocol from [21] which served as the inspiration for the key exchanges used in many protocols including IKE, OTR, and JFK.

In SIGMA', the initiator A first sends a message m_1 to B, consisting of Diffie-Hellman public value g^x, along with some protocol-specific parameters $info_A$ that may include, for example, a nonce, a protocol version, a proposed ciphersuite, etc. B responds with a message m_2 containing its own Diffie-Hellman public value g^y and some parameters $info_B$. A and B have now completed an anonymous Diffie-Hellman exchange and can compute the shared secret g^{xy} and use it to derive the session key. However, before using the session key, they authenticate each other by exchanging digital signatures over the protocol transcript $\mathsf{hash}(m_1|m_2)$ using their long-term signing keys (sk_A, sk_B). (Digital signature algorithms typically hash their arguments before signing them, and we have chosen to make this hashing explict in our presentation of SIGMA'.) By signing the transcript, A and B verify that they agree upon all the elements of the key exchange, and in particular, that a network attacker has not tampered with the messages. Finally, A and B also prove to each other that they know the session key g^{xy} by exchanging MACs computed with this key over their own identities.

Like other AKE protocols, SIGMA' aims to prevent message tampering, peer impersonation, and session key leakage, even if the network and other clients and servers are under the control of the adversary. Formally, authenticating the transcript guarantees *matching conversations*, that is, that the two parties agree on each others identity and other important protocol parameters.

Transcript Collision Attacks The alert reader will notice that SIGMA' *does not* in fact guarantee that A and B agree on the message sequence $m_1|m_2$; it only guarantees that they agree on the *hash* of this sequence. What if a network attacker were to tamper with the messages, so that A and B see different message sequences but the hashes of the two sequences is the same? In that case, the protocol will proceed to completion but the integrity and authentication guarantees no longer hold.

Figure 2 illustrates such an attack. The man-in-the-

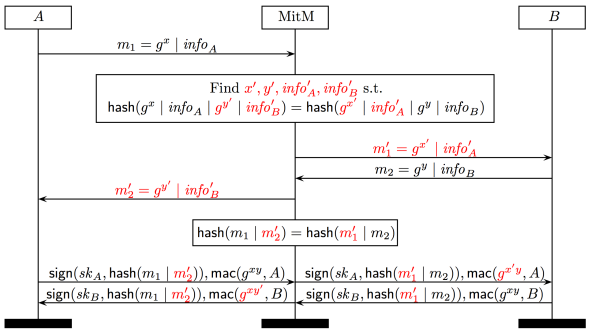

Fig. 2. Man-in-the-middle credential forwarding attack on SIGMA'. The attacker creates a *transcript collision* by tampering with the messages shown in red. At the end of the protocol, the client and server have seemingly authenticated each other, but the attacker knows both connection keys, and hence can read or write any data.

middle (MitM) intercepts messages sent between A and B. It sends its own message $m'_1 = g^{x'} | info'_A$ to B and it sends it own response $m'_2 = g^{y'} | info'_B$ to A. Suppose it can choose these messages such that the authenticated transcripts match:

$$\mathsf{hash}(m_1|m'_2) = \mathsf{hash}(m'_1|m_2)$$

We call this a *transcript collision*. Now, MitM can simply forward A's signature over this transcript to B and vice versa. A and B will accept the signatures since the hashed transcripts match and the signing keys are correct. However, the MitM knows the session keys $(g^{x'y}, g^{xy'})$ on both connections (since it knows x', y'). Hence, the MitM has fully hijacked both connections and can now send messages to B pretending to be A and to A pretending to be B. This is an *impersonation* attack that breaks peer authentication.

If the boundaries between the messages m_1 and m_2 are not clearly demarcated, there are a number of trivial attacks that can ensure that $m_1|m'_2 = m'_1|m_2$ with no need for hash collisions. In the examples of this paper, we will assume that each message (and each message field) is prefixed with its length, so that we can focus on attacks that rely on weaknesses in the hash function.

A Generic Transcript Collision The main challenge in implementing the attack in Figure 2 is that the MitM has to compute the messages m'_1 and m'_2 *after* receiving m_1 but *before* the responder has sent its response m_2. The feasibility of the attack depends on the contents and formats of these messages.

Suppose the responder B always sends the same message m_2 for every request; that is, it uses the same (static) Diffie-Hellman value g^y and same parameters $info_B$. (This situation occurs, for example in protocols like QUIC, where the server uses a static configuration.) In that case, after receiving m_1, the MitM can compute a transcript collision by finding $x', y', info'_A, info'_B$ such that $\mathsf{hash}(m_1|m'_2) = \mathsf{hash}(m'_1|m_2)$. The amount of work required to find such a collision depends on the hash function. As we will see in the next section, such collisions require $2^{N/2}$ work for hash functions that produce N bits. Hence, for MD5, such a collision would require the MitM to compute 2^{64} MD5 hashes, which may well be achievable by powerful adversaries.

A Chosen-Prefix Transcript Collision We now consider a more efficient attack that works even when B sends an unpredictable m_2 containing a fresh (ephemeral) Diffie-Hellman value g^y and a previously unknown $info_B$. However, we assume that the length of m_2 (M) is fixed and known to MitM. Moreover, suppose that in the second message of SIGMA', $info_B$ is allowed to have arbitrary length and arbitrary contents. That is, even if $info_B$ has junk data at the end, A will accept the message. Specifically, suppose that $info_B = len_B|data_B$ where $data_B$ is opaque data that will be ignored by A. (We will see several examples of such "collision-friendly" messages in TLS, IKE, and SSH.) Finally, we assume that the hash function uses the Merkle-Damgård construction [29], [7], so that it obeys the length extension property: if $\mathsf{hash}(x) = \mathsf{hash}(y)$ then $\mathsf{hash}(x|z) = \mathsf{hash}(y|z)$. (Strictly speaking, this property only holds when the lengths of x, y are equal and a multiple of the hash function block size.)

Under all these conditions, MitM can compute a transcript collision by finding two collision bitstrings C_1, C_2 of L_1 and L_2 bytes respectively, such that:

$$\mathsf{hash}(m_1|\underbrace{[g^{y'}|\overbrace{len'_B|C_1|-}^{info'_B}]}_{m'_2}) = \mathsf{hash}(\underbrace{[g^{x'}|\overbrace{C_2}^{info'_A}]}_{m'_1})$$

where $len'_B = L_1 + M$. Note that we have left empty space (written $-$) of size M bytes that still needs to be filled after C_1 in $info'_B$. As we will see in the next section, this kind of collision is called a *chosen-prefix* collision and is typically achievable with far less work than a generic collision attack. For example, a chosen-prefix collision in MD5 requires the MitM to compute about 2^{39} MD5 hashes, which takes only a few CPU hours.

After receiving m_1 from A and computing C_1, C_2, MitM now sends m'_1 to B. When B responds with m_2 (of size M bytes), MitM now stuffs m_2 at the end of $info'_B$ (in place of $-$) and sends m'_2 to A. Due to the length extension property, we have:

$$\mathsf{hash}(m_1|\underbrace{[g^{y'}|\overbrace{len'_B|C_1|m_2}^{info'_B}]}_{m'_2}) = \mathsf{hash}(\underbrace{[g^{x'}|\overbrace{C_2}^{info'_A}]}_{m'_1}|m_2)$$

That is, the MitM has obtained a transcript collision and the impersonation attack succeeds.

The attack here exploits hash collisions in combination with flexible protocol-specific message formats, and as we will see, this is one of the main novel *tricks* that we use to mount various attacks in this paper.

Other Transcript Collisions The transcript collisions described above are not the only attacks possible on such protocols. In some cases, MitM may not be able to use its own Diffie-Hellman values $g^{x'}, g^{y'}$ but it may still be able to tamper with the protocol parameters (e.g. ciphersuites) in $info'_A, info'_B$. In such cases, the MitM does not have full control over either connection (i.e. it cannot impersonate A or B) because it does know the session keys, but it may still be able to *downgrade* the protocol parameters to use weak, breakable ciphers.

In other cases, the message format may lend itself to simpler *common-prefix* collisions that require even less work than chosen-prefix collisions. Such collisions on MD5 can be found in seconds even on standard desktops. In the next section, we will discuss these different types of collisions in more detail (some technical details of previous results are given in the Appendix), and in the remainder of the paper, we will exploit them to mount transcript collision attacks on real-world protocols.

III. Hash Function Cryptanalysis

A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^N$ maps arbitrary length binary strings to strings of N bits. Broadly speaking, a cryptographic hash function is expected to behave like a randomly selected function from the set of all functions from $\{0,1\}^*$ to $\{0,1\}^N$; building input/output values with specific properties should be as hard for H as for a random function. More concretely, a cryptographic hash functions should meet four goals:

1) **Preimage resistance:** Given a target value $\overline{H}$, it should be hard to find x such that $H(x) = \overline{H}$
2) **Second-preimage resistance:** Given an input x, it should be hard to find a second input $x' \neq x$ such that $H(x') = H(x)$
3) **Chosen-prefix collision resistance:** Given prefixes P and P', it should be hard to find a pair of values x, x' such that $H(P'|x') = H(P|x)$.
4) **Collision resistance:** For a hash function H, it should be hard to find a pair of inputs $x \neq x'$ such that $H(x') = H(x)$.

The expected security of a hash function is defined as the complexity of the best generic attack, *i.e.* the best attack that works on any hash function, without using any specific property of the design. For preimages or second-preimages, the best attack is a brute-force search: an adversary has to try about 2^N random inputs in order to find a preimage. However, for collisions, there is a generic attack with complexity $2^{N/2}$ because of the birthday paradox. If an adversary computes the images of a set of $2^{N/2}$ inputs, this defines about 2^N pairs of inputs, and there is a high probability that one of these pairs is a collision.

Generic collision attacks While a naive collision attack requires to store $2^{N/2}$ images of the hash function, it is possible to mount a parallel and memory-less attack with a very small overhead [37]. This generic collision attack is very powerful: it can use meaningful messages, and can easily be used for chosen-prefix collisions (see details in Appendix).

Concatenation To strengthen protocols against collisions in any one hash function, it may be tempting to use a combination of two independent hash functions. For example, TLS versions up to 1.1 use a concatenation of MD5 and SHA-1. While the output length of this construction is 288 bits, it does not offer the security of a 288-bit hash function. In particular, Joux described a multi-collision attack that breaks the concatenation of two hash functions with roughly the same effort as breaking the strongest one of the two [18].

Shortcut collision attacks In the last decade, hash function cryptanalysis has been a very active research area, and more efficient attacks have been discovered on widely used hash functions. The (estimated) complexity of the best attacks currently known against MD5 and SHA-1 are the following:

MD5	Common-prefix collision: 2^{16} [36]
	Chosen-prefix collision: 2^{39} [36]
SHA-1	Common-prefix collision: 2^{61} [35]
	Chosen-prefix collision: 2^{77} [35]
MD5 | SHA-1	Common-prefix collision: 2^{67} [18]
	Chosen-prefix collision: 2^{77} [18]

Shortcut collision attack usually return messages with random-looking blocks that are not controlled by the adversary. This makes it harder to use these messages in a real attack, but we will see that in many cases we can still have meaningful messages by stuffing the random blocks in non-significant sections.

Implementation of attacks Since generic collision attacks can be easily parallelized and require little memory, they can efficiently be implemented in GPUs. In particular, an attack against MD5 require 2^{64} computations. This is well within reach for a motivated adversary: it would cost around $\$165\,000$ on Amazon EC2 (using a spot price of $8\,¢$/h for a g2.2xlarge instance doing $2.5\,$GH/s). Dedicated hardware would be significantly more efficient, but require a large investment. As

a point of comparison, the current Bitcoin network is able to compute up to 2^{59} SHA-256 hashes per second.

We have implemented this attack against the 96-bit MAC used for the Finished message of TLS 1.1. Our demo took 20 days using four Tesla K20Xm GPUs, which is comparable to the expected time we can derive from hash function benchmarks.

For a chosen-prefix collision, an important part of the computation is spend constructed differential paths, and this is much harder to parallelize on GPU. We used the HashClash software [33] by Marc Stevens to perform this computation. Stevens *et al.*'s estimate that the chosen-collision attack should require 2^{39} hash computations, or 35 core-hours [36]. In order to build the collision as fast as possible, we modified the software to take better advantage of parallelism. The hashclash software spends most of its time building differential paths, with a forward step, a backward step, and a connection step. We realized that the backward step uses a limited number of potential starting points, and we precomputed the results for all possible starting points. In addition, we merged the forward and connection steps, in order to avoid the serialization and deserialization of the result. With these optimisations, we can build a chosen-prefix collision in one hour with a 48 cores machine, using a few gigabytes of RAM (the original code required at least 3 hours). We believe the time can be further reduced, but this will require a significant rewrite of the hashclash software to allow parallelism across several machines, or to rewrite it for GPUs.

IV. The TLS Handshake Protocol

The Transport Layer Security protocol (TLS) [8] is perhaps the most widely used secure channel protocol. Many versions of TLS are used on the Internet; the latest released version is TLS 1.2 [8], while TLS 1.3 [9] is currently undergoing standardization at the IETF.

Figure 3 depicts a typical handshake in TLS (in versions 1.0 to 1.2). The client first sends a hello message `CH` that contains a fresh random client nonce n_c and various protocol parameters ex_c, including the protocol version, supported list of ciphersuites, and various protocol extensions. Each extension is prefixed by its length and can contain a payload of up to 2^{16} bytes. Notably, the client hello may include extensions that the server does not understand or support, and the server will ignore them.

The server responds to the client hello with a series of messages (from `SH` to `SHD`). The server hello `SH` contains a fresh server nonce n_s and parameters ex_s, including the server's chosen version, ciphersuite, and protocol extensions. In most ciphersuites, the server then sends its public-key certificate `SC`. In Ephemeral Diffie-Hellman (DHE) ciphersuites, `SC` is followed by a server

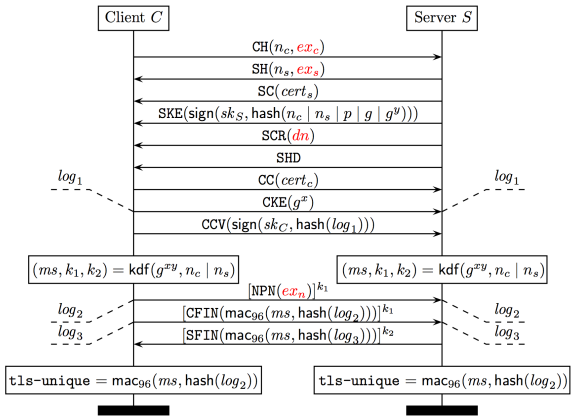

Fig. 3. TLS 1.2: A mutually-authenticated DHE handshake. Fields shown in red indicate parts of the handshake that can contain arbitrary-length opaque data (useful for stuffing collision blocks). Handshake transcripts (log_1, log_2, log_3) refer to the concatenation of all messages up to (and including) the current one. Messages `SCR`, `CC`, `CCV` are optional and only appear when client certificate authentication is used. `NPN` is optional and only appears when the client and server support the next-protocol-negotiation extension. The `tls-unique` channel binding is a connection identifier that may be used by applications to bind user authentication tokens, such as cookies and passwords, to the underlying TLS channel to prevent credential forwarding.

key exchange message `SKE` that contains an ephemeral public value g^y along with a description of the Diffie-Hellman group chosen by the server, including the prime p and generator g. The server signs these values to protect then from tampering and to prove that it knows the private key (sk_S) for the certificate:

$$\mathsf{sign}(sk_S, \mathsf{hash}(n_c|n_s|p|g|g^y))$$

The signature and hash algorithm used for this signature is chosen by the server based on its certificate as well as the supported algorithms indicated by the client within an optional `signature-algorithms` extension in the client hello. In TLS versions before 1.2, the hash algorithm was fixed to be MD5 | SHA-1 but TLS 1.2 allows clients and servers to choose any hash algorithm they both support (MD5, SHA-1, SHA-256, etc.) Hence in TLS 1.2, each digital signature is prefixed with identifiers for the chosen signature and hash algorithm.

If the server wants the client to authenticate itself with a public-key certificate, it then sends a certificate request message `SCR` indicating the certificate types and signature algorithms it supports, as well as an optional list of distinguished names dn for the client certification authorities that it trusts. As with hello extensions, each distinguished name can be 2^{16} bytes long and can contain arbitrary data that the client will ignore if it does not recognize the name. The server's message flight then ends with the server hello done message `SHD`.

The client then sends its own certificate `CC` if the

server asked for it, and its own Diffie-Hellman key share g^x in a client key exchange message CKE. If the client sent a certificate, it must prove that it knows the private key sk_C by sending a client certificate verify CCV message with a signature over the full message log up to this point in the protocol:

$$\mathsf{sign}(sk_C, \mathsf{hash}(\underbrace{\texttt{CH}|\texttt{SH}|\texttt{SC}|\texttt{SKE}|\texttt{SCR}|\texttt{SHD}|\texttt{CC}|\texttt{CKE}}_{log_1}))$$

At this point, the client and server both derive a session master secret ms and authenticated encryption keys for both directions (k_1, k_2). The client sends a change cipher spec message to indicate that the subsequent messages it sends will be encrypted (with k_1.) This message is not technically part of the handshake protocol and does not appear in the authenticated transcript, and so it is not shown in Figure 3.

If the client and server both indicate support for the next-protocol-negotiation extension [24] in their hello messages, the client then sends an encrypted extensions message NPN containing a selected application layer protocol (e.g. http/1.1 or spdy/3). The protocol name is ASCII-encoded and then padded to the nearest multiple of 32 bytes (to avoid leaking information via the encrypted message length.)

The client then sends an encrypted finished message CFIN containing a MAC of the full handshake log log_2 using the master secret ms. In TLS 1.0 and 1.1, this MAC is computed using a combination of HMAC-MD5 and HMAC-SHA-1, whereas in TLS 1.2, it uses HMAC-SHA-256. In all these versions, the result of the MAC is then truncated to 12 bytes (96 bits):

$$\mathsf{mac}_{96}(ms, \mathsf{hash}(\underbrace{\texttt{CH}|\texttt{SH}|\texttt{SC}|\texttt{SKE}|\texttt{SCR}|\texttt{SHD}|\texttt{CC}|\texttt{CKE}|\texttt{CCV}|\texttt{NPN}}_{log_2}))$$

When a server receives CFIN, it verifies that the client agrees with it on the full message log and on the master secret. It responds by sending its own change cipher spec message to turn on encryption and a server finished message SFIN that contains a 96-bit MAC over the full handshake log log_3 using the master secret ms.

At the end of the handshake, both client and server have authenticated each other, proved knowledge of the master secret, and agreed upon the message log. They can now start encrypting application data to each other using the connection keys (k_1, k_2).

In most common TLS usage scenarios, clients are not authenticated using certificates. The handshake authenticates only the server and the client-side user is authenticated within the application using a challenge-response protocol based on a password or some other bearer token (e.g. HTTP cookie). Such application-level authentication protocols are known to be vulnerable to a general class of *credential forwarding* attacks unless the application-level credential is channel bound to the TLS connection (e.g. see [5]. In such attacks, a client C connects to a malicious server M and authenticates with some credential over TLS, but M forwards the authentication message over another TLS channel to S, thereby logging in as C at S. The attack is prevented if the authentication protocol embeds a unique identifier for the underlying TLS channel, so that a message sent over one channel cannot be forwarded over another. One such identifier, called tls-unique, defined in [2], uses the contents of the CFIN message as a unique identifier for the TLS connection. This tls-unique channel binding is used by a number of emerging application-level authentication protocols, such as SCRAM [28], FIDO [14], and Token Binding [32], specifically to avoid credential-forwarding attacks.

V. Transcript Collision Attacks on TLS

As we saw in the previous section, TLS uses a variety of hash constructions to implement key security mechanisms like client and server authentication, handshake integrity, and channel binding. We now demonstrate weaknesses in these constructions and show how they can be exploited to mount practical transcript collision attacks on real-world clients and servers.

A. Breaking TLS 1.2 Client Authentication using a Chosen-Prefix Transcript Collision

Suppose a client C uses the same certificate to connect to two different servers A and S. We show that if A is malicious, it can force C to create a signature (in CCV) that A can use to impersonate C at S, as depicted in Figure 4. Here, A acts as a man-in-the-middle between C and S. Note, however, that A uses its own certificate $cert_a$ and does not rely on knowing any long-term secrets belonging to C or S.

Recall that the client signs the transcript $\mathsf{hash}(log_1^c)$; so the key idea of the attack is to compute a collision between this client-side transcript and the server-side transcript $\mathsf{hash}(log_1^s)$, even though the two connections see different message sequences. When the MitM A receives a client hello from C, it responds with its own hello SH′, certificate SC′, key exchange SKE′. It then initiates a connection with the server S by sending a carefully crafted client hello CH′. A now runs both connections in parallel. It will receive a hello SH, certificate SC, key exchange SKE, and certificate request SCR from S. We assume that the length of these messages SH|SC|SKE|SCR is fixed ($= M$) and is known in advance.

Note that A needs to choose CH′ before it receives any messages from S. A can compute CH′ and SCR′ as follows. A uses a chosen-prefix collision to find two bitstrings (C_1, C_2) of length L_1 and L_2 bytes respectively

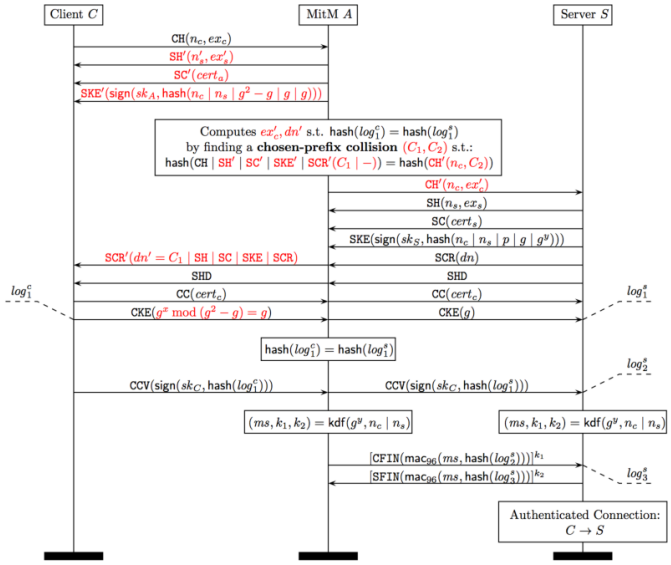

Fig. 4. Man-in-the-middle client signature forwarding attack on TLS 1.2. The client C connects to a malicious server A and offers to authenticate with its certificate $cert_C$. The attacker A computes a chosen-prefix collision on the client signature transcript $\mathsf{hash}(log_1)$, and uses it to impersonate the client at a different server S. Messages that the attacker controls are labeled in red. A sends a bogus Diffie-Hellman group $(k^2 - k, k)$ to C; we use $k = g$ here for simplicity.

such that C_1 appears within the last distinguished name dn' in $\mathtt{SCR}'$ and C_2 appears within the last extension ex'_c in $\mathtt{CH}'$:

$$\begin{aligned} & \mathsf{hash}(\mathtt{CH}|\mathtt{SH}'|\mathtt{SC}'|\mathtt{SKE}'|\mathtt{SCR}'(\underbrace{C_1}_{dn'}|-)) \\ = \; & \mathsf{hash}(\mathtt{CH}'(n_c, \underbrace{C_2}_{ex'_c})) \end{aligned}$$

Furthermore, we set the length of dn' in $\mathtt{SCR}'$ to be $L_1 + M$, so that it still has M bytes (denoted by $-$) that need to be filled in after C_1.

Now, A sends $\mathtt{CH}'$ to S, receives $\mathtt{SH}|\mathtt{SC}|\mathtt{SKE}|\mathtt{SCR}$ in response, and stuffs these messages into the remaining M bytes in $\mathtt{SCR}'$ and sends it to C. At this point the hash of the message transcripts in the two connections coincide, assuming that the hash function satisfies the length extension property:

$$\begin{aligned} & \mathsf{hash}(\mathtt{CH}|\mathtt{SH}|\mathtt{SC}'|\mathtt{SKE}'|\mathtt{SCR}'(\underbrace{C_1|\mathtt{SH}|\mathtt{SC}|\mathtt{SKE}|\mathtt{SCR}}_{dn'})) \\ = \; & \mathsf{hash}(\mathtt{CH}'(n_c, \underbrace{C_2}_{ex'_c})|\mathtt{SH}|\mathtt{SC}|\mathtt{SKE}|\mathtt{SCR}) \end{aligned}$$

From this message onwards, the hash of the handshake log in both connections will remain the same. A then forwards the sever hello done $\mathtt{SHD}$ to C. In response, C sends a certificate $\mathtt{CC}$, a key exchange $\mathtt{CKE}$, and a certificate verify $\mathtt{CCV}$ that contains a signature over the transcript $\mathsf{hash}(log_1^c)$ which is now the same as $\mathsf{hash}(log_1^s)$. A simply forwards these messages to S, pretending to be C, and S accepts these messages.

Controlling the master secret Even though S has accept C's certificate on its connection with A, A cannot complete the connection unless it knows the master secret on its connection with S. The master secret is computed from g^{xy} so A needs to know the x corresponding the g^x that C sent in its key exchange message $\mathtt{CKE}$. In order to accomplish this task, we rely on a key forcing attack in the DHE handshake.

When A sends $\mathtt{SKE}'$ to C, it does not send a valid Diffie-Hellman group (p, g). Instead, it chooses an arbitrary public value $k = g^{x'}$ and sets $p = k^2 - k$ and $g = k$. This p value is clearly not a prime, and it has the property that no matter what private value x is generated by C, we will have $g^x \bmod p = k$. Hence, by choosing such a bogus Diffie-Hellman group, A can force C to send a $\mathtt{CKE}$ with a public value that it controls.

To complete the attack, we assume that S always uses the same Diffie-Hellman group (p, g). A chooses some x' and sets $k = g^{x'} \bmod p$. It then sends $\mathtt{SKE}'$ to C with the bogus group $(k^2 - k, k)$ and the public value k. Now, the $\mathtt{CKE}$ sent by C will contain k, and A will forward it to S. A will then forward C's signature $\mathtt{CCV}$ as usual. The master secret between A and S will be derived from $g^{x'y} \bmod p$, but A knows x' and hence can compute this value. Consequently, A can complete the handshake and impersonate C at S.

We observe that the attack here relies on the client not validating the Diffie-Hellman groups it receives from the server. From our experiments, we find that most TLS libraries do not validate the groups they receive in the server key exchange, probably because checking for primality is expensive. In some libraries, the value $k^2 - k$ is rejected because it is an even number. In those cases, we find that we can use $p = k^2 - 1$ and with 50% probability, the client will compute $g^x = k$, allowing the attack to succeed. This weakness in TLS-DHE has been noted before [6] and a new protocol extension aims to fix it by allowing only well-known Diffie-Hellman groups [12]. However, an optional extension cannot prevent our attack scenario, since A could always pretend to not support the extension and mount the attack anyway.

Note that the attack only relies on DHE between C and A; the connection between A and S can use ECDHE or RSA and the attack would still work. In other words, such transcript collisions can also be used to mount cross-protocol attacks in the sense of [26].

Attack Complexity The transcript collision attack requires A to compute a chosen-prefix collision for the hash function used in the client signature. In TLS

versions before 1.2, the default hash function is a concatenation of MD5 and SHA-1 and hence requires computing 2^{77} MD5 and SHA-1 hashes. In TLS 1.2, if the signature uses SHA-1, the cost is 2^{77} hashes. Remarkably, TLS 1.2 also allows RSA-MD5 signatures, and for such signatures, the cost of the collision is only 2^{39} MD5 hashes. Below, we describe our proof-of-concept implementation that relies on RSA-MD5.

Note that these cost estimates are per-connection because the collision needs to be computed once for each client nonce n_c. Usually, these nonces are generated with a strong random number generator. However, in some cases the client random can become predictable due to implementation bugs (e.g. see CVE-2015-0285 in OpenSSL). We also observe that it is commonly believed that these nonces only need to be unique, not unpredictable. For example, the OpenSSL library uses `RAND_pseudo_bytes` to generate the client and server random, whereas it uses `RAND_bytes` to generate other key material; the former succeeds even when there is not enough entropy in the system. If the client nonce were predictable, or if it were to be repeated with high frequency, the collision can be computed offline at leisure, making SHA-1 collisions almost feasible. Even though our attack below does not rely on predictable nonces, it offers yet another justification for the need for strongly random nonces in TLS.

Implementing a Proof-Of-Concept To implement the attack, we need a client that is willing to sign with RSA-MD5 and a server that is willing to accept such signatures. We found a number of TLS libraries that support RSA-MD5 client signatures, including certain versions of OpenSSL, GnuTLS, Oracle and IBM Java, and BouncyCastle. (See Section VII for more details.) In particular, all major Java web application servers and the default TLS servers on Red Hat Enterprise Linux (6 and 7) accept RSA-MD5 signatures.

For our demo, we set up a man-in-the-middle attack between a standard Java HTTPS client and a Java HTTPS server (with default configurations.) The MitM implements Figure 4. In order to setup the collision while preserving the TLS message formats, the attacker needs to carefully set the length fields in various places in $\texttt{CH}'$ and $\texttt{SCR}'$. For example, in $\texttt{CH}'$ it needs to set consistent lengths for the full hello message, for the extensions field, and for the last extension. Furthermore, the MitM needs to make sure that the two prefixes have a length that is a multiple of the MD5 block size (512 bits). To achieve this, we fill up the last extension in $\texttt{CH}'$ and the last distinguished name in $\texttt{SCR}'$ with enough zero bytes until the prefixes are block-aligned.

As explained in Section III, the chosen-prefix collision can be be computed in one hour on a 48 core workstation using a modified version of the hashclash software [33]. In our demo, A accepts the client hello and then keeps the client-side TLS connection alive until a collision has been found. Most TLS connections can be kept alive by sending regular warning alerts; Java clients are willing to keep the connection open indefinitely. Keeping the client waiting for an hour is not always practical, but we note that some unsupervised TLS clients (such as git) are used to perform long-running connections to web APIs, and long connection times may not be noticed. In any case, the collision search scales well with computational power and can be significantly sped up by a powerful adversary.

Once the collision has been found, A connects to S to completes the attack and is able to impersonate C at S and read and write data that only C should have access to. Hence, the demo shows that A is able to break TLS 1.2 client authentication between mainstream TLS clients and servers. The precise handshake traces exhibiting the collision are available from our website.

B. Breaking TLS 1.2 Server Authentication using a Generic Transcript Collision

The key to our attack above on TLS 1.2 client authentication is that the client is willing to sign the hash of the full message log, and the format of various TLS messages is flexible enough to allow the attacker to stuff meaningless collision blocks and server-side messages into them. A similar chosen-prefix transcript collision attack would not work on TLS 1.2 server authentication because the server signature transcript does not contain flexible-size elements.

In DHE handshakes, the signature covers only the client and server nonces and the server's Diffie-Hellman key share: $\mathsf{sign}(sk_S, \mathsf{hash}(n_c|n_s|p|g|g^y))$. So, the only part of the signed value that the attacker may control is the client nonce n_c which is fixed-length (32 bytes), half the size of one MD5 block.

This prevents the use of shortcut collision attacks against MD5, but generic attacks based on the small 128-bit MD5 hash length are still possible, and not too far from being practical.

Collecting and storing signatures. To mount a transcript collision attack on TLS 1.2 server authentication, an attacker first has to collect a large number, say 2^x, of RSA-MD5 signatures signed by the server. The attacker may do this by passively observing RSA-MD5 connections to the server, but since such connections may be rare, it may have to actively connect to the server to obtain a sufficient number of signatures. Once these signatures (and the corresponding hashes) have been collected and stored, the attacker can impersonate the server to any client.

Upon receiving a client hello message including the client nonce n_c, the attacker chooses a DH secret y' and computes the MD5 hashes of the transcripts $n_c|n_s|p|g|g^{y'}$ for a series of random server nonces n_s, until the hash matches a value that was collected previously. Finding this collision requires the attacker to compute about 2^{128-x} MD5 hashes and then look them up in the stored signature database. When a match is found, the stored signature can be used by the attacker to forge the server's SKE message for the current connection, and hence impersonate the server.

The complexity of this attack on TLS 1.2 server authentication is therefore 2^{128-x} MD5 hashes per connection, in addition to 2^x connections performed before-hand, and 2^x storage. The attacker can trade-off between these costs—the more signatures he can collect, the less he has to compute per connection. For example, if it is feasible to collect, store, and search through 2^{64} signatures, then the per-connection cost is 2^{64} hashes. Although we have described the attack in terms of MD5, a similar but more expensive attack can be mounted on RSA-SHA1 server signatures, which would require 2^{160-x} computation per-connection.

Practical Impact of the Attack. Both the precomputation and per-connection cost of the attack is currently out of reach for academic researchers, but might be within the capabilities of well-resourced adversaries.

A prerequisite for the attack is to find servers that would be willing to sign their SKE messages with RSA-MD5. Internet-wide scans show that about 31% of the Alexa top 1 million websites support RSA-MD5 signatures.[1] This subset includes popular websites hosted by Akamai, such as microsoft.com.

A second question is whether TLS clients would accept RSA-MD5 signatures. Most popular web browsers and TLS libraries do not offer RSA-MD5 as one of the supported signature algorithm in the client hello. This might lead one to believe that they would not accept RSA-MD5 server signatures. However, we found and reported security bugs in NSS (the library used by Firefox and some versions of Chrome), GnuTLS (used in curl and git), and BouncyCastle; these libraries (and applications that rely on them) incorrectly accept RSA-MD5 signatures even if they have been explicitly disabled. For example, Firefox will accept an RSA-MD5 signature from a website, even though it is not supposed to. Furthermore, other TLS libraries such as versions of OpenSSL (up to version 1.0.1e), mbedTLS, and Java routinely offer and accept RSA-MD5 signatures.

Conseqently, if an attacker has the resources to achieve the server impersonation attack, a large number of TLS clients would be affected. To err on the safe side, we recommend that TLS libraries should immediately disable all MD5-based signatures.

[1] https://securitypitfalls.wordpress.com/2015/12/07/november-2015-scan-results/

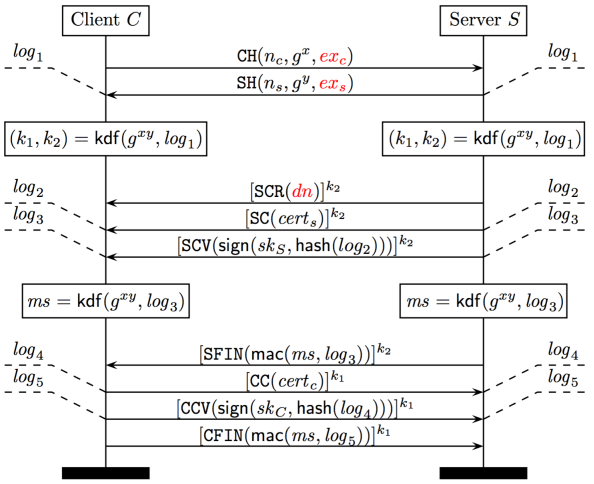

Fig. 5. TLS 1.3: A server-authenticated 1-RTT (EC)DHE handshake based on draft 10 of the specification. The client and server send their key shares within the hello messages and all subsequent handshake messages are encrypted. The server signs the current handshake transcript $\mathsf{hash}(log_2)$ in a new SCV message.

Exploiting predictable nonces and keys. We observe that the precomputation in the above attack can be avoided if the server uses a predictable nonce n_s and predictable DH parameters p, g, g^y. In this case, the attacker only has to perform 2^{64} computations online. How realistic is this assumption? Many TLS implementations allow DH keys to be reused; in OpenSSL, for example, keys are reused by default unless the application sets the SSL_OP_SINGLE_DH_USE flag. For such servers, the parameters are clearly predictable.

That leaves the server nonce, and as we noted for clients, a bug in the use of the random number generator could lead to predictable nonces. Alternatively, the server may support a recent TLS variant called Snap-Start [23] that allows the client (and hence the attacker) to choose the server nonce. That said, we do not know of any deployed TLS 1.2 implementation that uses predictable nonces, but this section serves as a warning to implementors that strongly random nonces are needed in TLS, and not just for preventing replays.

C. Breaking TLS 1.3 Client and Server Authentication using a Chosen-Prefix Transcript Collision

From the viewpoint of transcript collisions, TLS 1.2 server signatures may seem stronger than client signatures, but not signing enough leads to other security problems. For example, the server becomes vulnerable

to cross-protocol attacks [26] and to downgrade attacks like Logjam [1]. In response to such attacks, the new design of TLS 1.3 requires the server to sign the full handshake log, including the negotiated parameters.

Figure 5 illustrates the standard one-round-trip (1-RTT) message flow in the current draft (version 10) of the TLS 1.3 specification. In comparison to TLS 1.2, this protocol flips the order in which the DH key shares are sent, so that the handshake can complete in one round trip. The key shares are sent within extensions in the hello messages CH and SH. The server no longer sends an SKE message. Instead, it sends a new server certificate verify message SCV just before the finished message. The SCV contains a signature over the hash of the full message log up to this point (log_2). Another departure from TLS 1.2 is that all handshake messages after SH are encrypted, in order to protect the privacy of the client and server certificates from passive attackers.

We demonstrate a chosen-prefix transcript collision on TLS 1.3 that breaks both client and server signatures, enabling a full man-in-the-middle attack on the protocol. The attack is similar in spirit to the one on TLS 1.2 client signatures; we use the flexible formats of the client and server hello messages to create a trascript collision immediately after the server hello SH.

The client C wants to connect to S, but its messages are intercepted by a network attacker A. After A receives the client's CH, it sends its own $\texttt{CH}'$ to the server, receives the servers SH, and sends its own $\texttt{SH}'$ to the client. A now knows the Diffie-Hellman shared secrets on both connections, and it has chosen $\texttt{CH}'$ and $\texttt{SH}'$ such that $\mathsf{hash}(\texttt{CH}|\texttt{SH}') = \mathsf{hash}(\texttt{CH}'|\texttt{SH})$. Consequently, A can now simply forward all hanshake messages between C and S, and both client and server authentication will succeed. A will need to decrypt and reencrypt these messages, but it can do so because it knows the encryption keys on both connections. More importantly, once the handshake is complete, A can read and tamper with application data in both directions.

To compute $\texttt{CH}'$ and $\texttt{SH}'$, A needs to find a chosen-prefix collision C_1, C_2 of length L_1 and L_2 bytes respectively such that C_1 appears within the last extension of of $\texttt{SH}'$ and C_2 appears as the last extension of $\texttt{CH}'$:

$$\mathsf{hash}(\texttt{CH}|\texttt{SH}'(n_s, g^{y'}, \underbrace{C_1|-}_{ex'_s})) = \mathsf{hash}(\texttt{CH}'(n_c, g^{x'}, \underbrace{C_2}_{ex'_c}))$$

Suppose we know that the server S will respond to $\texttt{CH}'$ with a server hello message SH of known length M. Then in $\texttt{SH}'$, we set the length of ex'_s to $L_1 + M$ so that there is room for M more bytes after C_1. Once A receives SH from S, it stuffs this message within this extra space in $\texttt{SH}'$ and sends it to C. Hence, after the server hello, the handshake transcripts at the client and server have the same hash. Moreover, due to the length extension property of the hash function, all subsequent handshake hashes collide. So, A can forward S's signature in SCV to C and C's signature in CCV to S, and both will be accepted, even though the DH keys have been tampered with by a man-in-the-middle.

Implementing a Proof-Of-Concept Up to draft 7, the TLS 1.3 specification explicitly allowed RSA-MD5 signatures. We wrote a proof-of-concept attack demo based on our own simple prototype implementation of TLS 1.3 that signs with RSA-MD5. As with TLS 1.2 client authentication, we found the chosen-prefix collision in roughly one hour on a single workstation.

As we observed when discussing TLS 1.2, a large number of TLS servers and clients support RSA-MD5 signatures. Consequently, we believe that if TLS 1.3 draft 7 were to be implemented today, it is quite likely that many of its clients and servers would be vulnerable to our man-in-the-middle attack. However, this attack vector was removed from TLS 1.3, at least partly due to our findings, when draft 8 of the protocol explicitly deprecated MD5-based signatures.

D. Downgrading TLS 1.0-1.1 to Weak Ciphersuites using a Chosen-Prefix Transcript Collision

In TLS, the integrity of the handshake depends upon the MACs exchanged in the Finished messages. If these MACs were broken, the attacker would be free to modify the hello messages to downgrade the connection to an old protocol version or weak ciphersuite, or to delete important extensions such as the renegotiation indication countermeasure [11].

Recall that the Finished MACs are computed over the *hash* of the full handshake trascript ($\mathsf{hash}(log_2)$ and $\mathsf{hash}(log_3)$ in Figure 3). In TLS 1.0 and 1.1, this hash function is the concatenation of MD5 and SHA-1. As we saw in Section III, a chosen-prefix collision on this construction can be computed with 2^{77} work. We find a man-in-the-middle transcript collision attack on server-authenticated TLS 1.1 that is similar to the TLS 1.3 attack. A network attacker modifies the client and server hellos so that the handshake hashes collide immediately after these two messages; the rest of the handshake is left unchanged. The client authenticates the server and the handshake completes successfully, and although the attacker does not know the master secret, it can downgrade the connection to use any weak algorithm that both the client and server support, but prefer not to use, such as an EXPORT ciphersuite [1], or a weak encryption algorithm like RC4.

A similar transcript collision attack appears in DTLS 1.0, a UDP-based variant of TLS 1.1. In DTLS, the attack can be made even more efficient by exploiting its cookie mechanism. In response to a client hello CH,

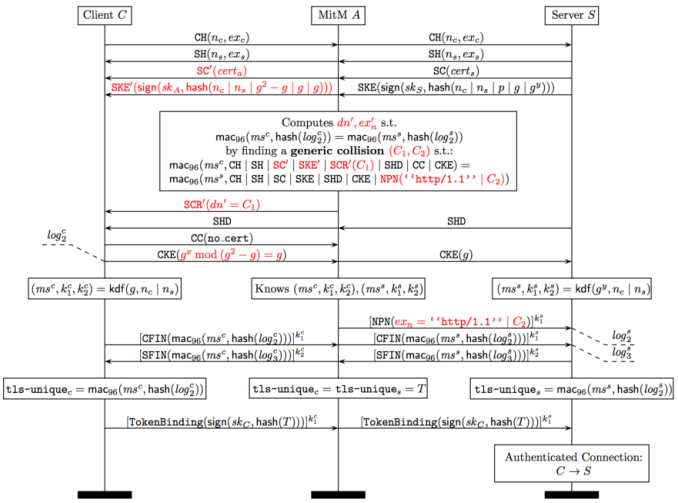

Fig. 6. Man-in-the-middle credential forwarding attack on `tls-unique` channel binding. The attack uses a transcript collision to impersonate the client to the server.

a DTLS server can send a hello verify request message `HVR` containing a cookie. The client is meant to restart the handshake by sending the exact same client hello message but with this cookie included in it. Since the `HVR` is not authenticated, the arbitrary-length cookie field allows any network attacker to inject data into the transcript, after a known prefix of a fixed length. This allows the chosen-prefix attack to be transformed to an *almost* common prefix attack, similar to the cookie-based attack on IKEv2 in Section VI-A.

E. Breaking the `tls-unique` Channel Binding using a Generic Transcript Collision

Suppose an application-level authentication protocol at C binds its login credential to the `tls-unique` channel binding [2], so that when the credential is sent from C to A, it cannot be used by A at S. We demonstrate how the attacker A could use a generic collision attack to break this protection.

Figure 6 depicts the attack. It follows the general pattern of the TLS 1.2 client authentication attack, except that it relies on a collision on the transcript MAC in the client finished message, rather than a collision in the hash function. The client C connects to the MitM A who then opens a new connection to S. The attacker sends a `SKE`′ to C that contains a bogus group $(k^2 - 1, k)$, thereby forcing the client to send $k^x \bmod (k^2 - 1) = k$ in its client key exchange `CKE`. On the server side, the attacker can send its own `CKE`′ containing any Diffie-Hellman value. Hence, the MitM knows the master secrets ms^c, ms^s and connection keys on both connections.

The goal of the attacker is to make sure that the contents of the client finished message (i.e. the `tls-unique`) coincide on both connections:

$$\mathsf{mac}_{96}(ms^c, log_2^c) = \mathsf{mac}_{96}(ms^s, log_2^s)$$

The attacker can use any controlled part of the transcript, but we will set things up carefully so that he can compute the collision as late as possible, in order to reduce the size of the messages to hash. More precisely, we use the certificate request `SCR`′ on the client-side and the `NPN` message on the server side, which are sent when all other messages in the transcript are already fixed. The attacker uses C_1 as the last distinguished name in `SCR`′ and C_2 as the padding in the `NPN` message (after the protocol name "`http/1.1`"), and computes (C_1, C_2) such that the MAC coincides. Once this collision is found, the MitM sends these two messages on the corresponding connections and completes the handshakes. A can then impersonate C at S by forwarding any application-level channel-bound credentials sent by C (for A) to S.

Implementing a Proof-Of-Concept We implemented a man-in-the-middle attacker to demonstrate the attack. We used an OpenSSL client as C and the main Google website as S, since this website supports the next-protocol-negotiation protocol extension. After receiving the client hello `CH` from the client and the server hello done `SHD` from the server-side, the MitM runs a generic collision search to compute `SCR`′ and `NPN`.

For the collision search, we implemented the TLS PRF mac_{96} function using the CUDA framework for NVIDIA GPUs. In TLS versions up to 1.1, this construction is built using MD5 and SHA-1; in TLS 1.2 the construction uses SHA-256. However, the strength of the hash function is immaterial because what we are attacking is the truncated 96-bit MAC. The underlying hash function does not matter. Following the analysis explain in Section III, it should require about 2^{48} computations on average to get a collision.

Our implementation run at 160 MH/s for TLS 1.1 and 113 MH/s for TLS 1.2 on a Tesla K20Xm GPU. This is comparable to the expected speed we can derive from benchmarks of MD5, SHA-1 and SHA-256 on this GPU.It took 20 days to find a collision for TLS 1.1, using four Tesla K20Xm GPUs. Our demo evaluated the PRF about $2^{49.9}$ times, which is rather unlucky: it should take half that number on average. We note that the generic collision attack is completely parallelizable and hence the time for finding a collision can be brought down to an arbitrarily small number by throwing enough computational power at it. Using Amazon EC2, this should cost about $140 for TLS 1.1, and $200 for TLS 1.2. The transcripts are available on our website.

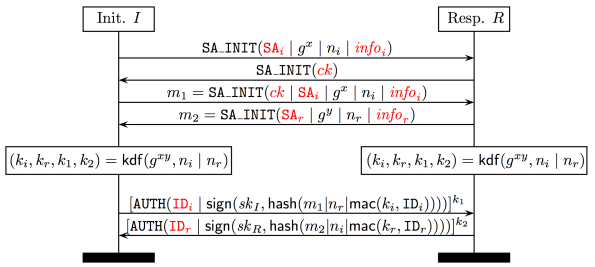

Fig. 7. IKEv2: A mutually-authenticated key exchange. Message parts colored in red can have arbitrary length.

Truncated HMAC is not collision-resistant A more general lesson to be taken from our attack on `tls-unique` is that there are many uses of HMAC in cryptographic protocols that are not protected from collisions in the underlying hash function. For example, although HMAC-MD5 may be a good MAC, it is not collision-resistant when the key is known to the attacker. Similarly, when HMAC-SHA256 is truncated to 96 bits, it may still be a good MAC, but it is certainly not a good hash function (since collisions can still be found in 2^{48} steps). Consequently, when inspecting the use of hash functions in Internet protocols, it would be a mistake to assume that all uses of HMAC are safe; it is important to look both at the mechanism and its intended security goal. In some cases, we may need HMAC to be both a MAC and a collision-resistant hash function.

VI. Transcript Collisions in IKE and SSH

Although the bulk of this paper has focused on collisions in TLS, similar attacks apply to other mainstream protocols like IKEv1, IKEv2, and SSH. We describe two exemplary attacks here.

A. Breaking IKEv2 Initiator Authentication using a Precomputed Common-Prefix Transcript Collision

Figure 7 depicts the IKEv2 authenticated key exchange protocol, which is similar to the SIGMA' protocol discussed in Section II. The initiator first sends an `SA_INIT` request containing its Diffie-Hellman value g^x, nonce n_i, and proposed cryptographic parameters $\mathtt{SA}_i, info_i$. The responder replies with its own public value g^y, nonce n_r and parameters $\mathtt{SA}_r, info_r$. Alternatively, the responder may send a cookie ck, thereby asking the initiator to restart the protocol by sending the same `SA_INIT` request but with ck included in it.

After the `SA_INIT` exchange, the initiator and responder authenticate each other by signing a portion of the message transcript. Notably the initiator signs:

$$\mathsf{hash}(\underbrace{\mathtt{SA\_INIT}(ck|\mathtt{SA}_i|g^x|n_i|info_i)}_{m_1}|n_r|\mathsf{mac}(k_i, \mathtt{ID}_i))$$

Figure 11 in Appendix depicts an attack on IKEv2 initiator authentication that relies on a transcript collision on this signature. The network attacker intercepts the `SA_INIT` request from I to R and responds with a cookie ck. The initiator I restarts the key exchange by including ck in the new `SA_INIT` request (m_1). However, the attacker has chosen ck in a way that the hash of m_1 is the same as the hash of a tampered `SA_INIT` request m_1' that contains the attacker's Diffie-Hellman public value $g^{x'}$. The attacker sends this tampered request m_1' to the responder and upon receiving a response, it tampers with the response to replace R's Diffie-Hellman key g^y with its own key $g^{y'}$. Note that the attacker does not tamper with the nonces n_i, n_r.

At this point, the attacker knows the shared secrets $g^{x'y}, g^{xy'}$ and encryption keys on the two connections. Moreover the hash used in the signature transcript collides all the way to the $\mathsf{mac}(k_i, \mathtt{ID}_i)$. To complete the attack, the attacker must ensure that k_i is that same at I and R. It can ensure this by choosing x', y' such that $g^{x'y} = g^{xy'}$ (as discussed below). Thereafter, it can forward I's signature to R and hence impersonate I.

Implementing the Attack To implement the attack, we must first find a collision between m_1 and m_1'. We observe that in IKEv2 the length of the cookie is supposed to be at most 64 octets but we found that many implementations allow cookies of up to 2^{16} bytes. We can use this flexibility in computing long collisions.

The attacker finds two length-prefixed bitstrings (C_1, C_2) of L bytes each such that

$$\mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{[C_1|-]}_{ck}|-)) = \mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{C_2}_{ck'}|-))$$

where the length of ck is set to $L+M$, that is, ck has M empty bytes ready to fill in. We set M to the length of the bitstring $\mathtt{SA}_i'|g^{x'}|n_i$ that the attacker wants to send to R in its tampered `SA_INIT` request m_1'. The idea is that the attacker can now stuff the tampered message into ck, and can stuff the original message into $info_i'$ to obtain a transcript collision:

$$\mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{[C_1|\mathtt{SA}_i'|g^{x'}|n_i]}_{ck}|\mathtt{SA}_i|g^x|n_i|info_i)|-) =$$
$$\mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{[C_2]}_{ck}|\mathtt{SA}_i'|g^{x'}|n_i|\underbrace{[\mathtt{SA}_i|g^x|n_i|info_i]}_{info_i'})|-)$$

The collision (C_1, C_2) can be found easily as a chosen-prefix collision attack. Since the collision occurs before any unpredictable value has been included in the message, it can be computed offline; that is, it does not have to be computed while a connection is live. The collision can then be used to break any number of connections between I and R. Such collisions are easy to compute for MD5, but we found that even

though MD5 signatures are allowed by the standard, they are not commonly supported by IKEv2 implementations. However, SHA-1 signatures are mandatory for all IKEv2 implementations, so an offline chosen-prefix collision on SHA-1 is enough to mount the attack. The best known complexity of such collisions is currently 2^{77}, which may be feasible for a powerful adversary (especially if better shortcut attacks on SHA-1 are discovered).

We also observe that the two prefixes are very similar: we only need the length of the cookie to be different. Following the format of IKE message, the length field is on bytes 22 and 23 of the hashed transcript, and all previous bytes must have a fixed value. Hence, we can *almost* use a common-prefix collision attack, if the collision algorithm introduces a difference in bytes 22-23, and no difference in preceding bytes. For MD5, the most efficient collision attacks do not have a compatible message difference, but it seems possible to build a dedicated attack with complexity below 2^{39}. However, for SHA-1, all known collision attacks use differences in every message words, and are thus unsuitable.

The final step to implement the attack is to ensure that $g^{xy'} = g^{x'y}$. To achieve this, we rely on a small subgroup confinement attack. To see a simple example, suppose the attacker chose $x' = y' = 0$; then the two shared secrets would have the value 1. This specific solution would not work in practice because most IKEv2 implementations validate the received Diffie-Hellman public value to ensure that it is larger than 1 and smaller than $p - 1$. However, many IKEv2 implementations support the Diffie-Hellman groups 22-24 that are known to have many small subgroups. These implementations do not validate the incoming public value, and hence are susceptible to similar small subgroup confinement attacks, as discussed in [5]. To complete our transcript collision attack, the MitM can use one such small subgroup to ensure that the shared values on the two connections are the same with high probability.

B. Breaking IKEv1 Initiator Authentication with a Generic Transcript Collision

IKEv1, the predecessor of IKEv2, and is also vulnerable to transcript collision attacks. We briefly outline one attack, without giving more details for lack of space. The initiator's signature in IKEv1 is computed as:

$$\mathsf{sign}(sk_I, \mathsf{prf}(\mathsf{prf}(n_i|n_r, g^{xy}), g^x|g^y|c_i|c_r|\mathtt{SA}_i|\mathtt{ID}_i))$$

A commonly-used PRF function in IKEv1 is HMAC-MD5, and we find a generic transcript collision attack on the outer PRF value that allows initiator impersonation. A man-in-the-middle attacker intercepts a connection between I and R; it tries out many random $g^{y'}$ values on the client-side, and many random values (embedded in) $\mathtt{ID}'_i$ on the server side, until the PRF values on the two sides collide. It can then forward I's signature to R, even though it knows the Diffie-Hellman shared secret. When the PRF is HMAC-MD5, this generic collision attack costs about $2 * 2^{64}$ HMAC computations per connection.

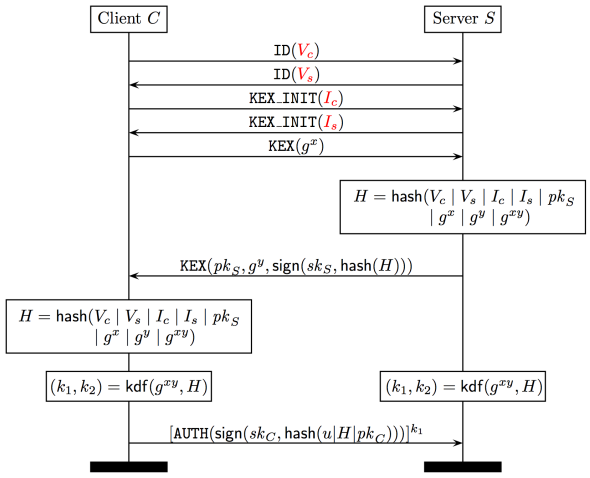

Fig. 8. SSH-2: Key exchange and user authentication.

C. Downgrading SSH-2 to Weak Ciphersuites with a Chosen-Prefix Transcript Collision

Figure 8 depicts the SSH-2 [39] protocol, which implements yet another variation of an authenticated Diffie-Hellman protocol. The client and server exchange identification strings V_c, V_s, negotiate protocol parameters I_c, I_s, and perform a Diffie-Hellman exchange g^x, g^y. To authenticate the exchange, clients and servers sign a session hash, defined as:

$$H = \mathsf{hash}(V_c|V_s|I_c|I_s|pk_S|g^x|g^y|g^{xy})$$

We show that a target collision on this hash value can allow downgrade attaks.

Figure 12 in Appendix depicts a downgrade attack on SSH-2. The network attacker tampers with the key exchange message I_c in one direction and with I_s in the other. It chooses their values in a way such that the following hashes coincide

$$\mathsf{hash}(V_c|V_s|I_c|\underbrace{C_1|-}_{I'_s}) = \mathsf{hash}(V_c|V_s|\underbrace{C_2}_{I'_c})$$

Using this collision, we leave enough space empty in I'_s to stuff the real I_s inside. Consequently the session hashes on the two sides coincide and the connection is completed. In this attack, the MitM does not tamper with the Diffie-Hellman values and hence it does not know the connection keys. However, it manages to

tamper with both I_c and I_s, and can therefore downgrade the negotiate ciphersuite to a weak cryptographic algorithm that the attacker knows how to break.

Implementing the target collision for SSH-2 requires a chosen-prefix attack on SHA-1 which is still considered impractical (at least 2^{77} work). Moreover, since the two tampered fields I_c and I_s are meant to be strings (not bitstrings), we cannot use arbitrary collisions. Still, we find this attack to be an interesting illustration of the use of transcipt collisions for downgrade attacks.

SSH-2 has a peculiar session hash construction, with the shared secret g^{xy} placed at the end. This makes certain kinds of collision attacks more difficult, but we note that this construction is not particularly secure; since it includes the shared secret, the session hash needs to be *non-leaking* in addition to being collision-resistant [4]. Moreover, if the SSH server reuses its Diffie-Hellman public value, this secret suffix becomes vulnerable to key recovery attacks like on APOP [25].

Other variations of SSH allow for more tampering, which may enable new attacks. The SSH Diffie-Hellman Group Exchange protocol [10] allows SSH servers to choose any Diffie-Hellman group for use in the key exchange. So, like in our TLS attacks, a man-in-the-middle attacker can send a bogus or weak group to the client, and use it to control more fields in the session hash and mount new transcript collision attacks.

VII. SLOTH: Responsible Disclosure and Impact

Table I summarizes the attacks discussed in this paper. Three of our attacks on TLS are already practical; others are within the reach of powerful adversaries.

Our attacks on TLS were publicly disclosed under the acronym SLOTH (security losses from obsolete and truncated transcript hashes) and were assigned a protocol-level CVE-2015-7575. We informed the authors of affected protocol specifications and developers for various TLS libraries. We recommended that protocols and implementations should stop using MD5-based signatures and other weak hash constructions. Our disclosure and recommendations resulted in the following security updates:

1) TLS 1.3 draft 7 stopped truncating the Finished MACs and started using the full HMAC output.
2) TLS 1.3 draft 8 deprecated MD5 signatures.
3) The Token Binding Protocol draft 2 removed `tls-unique` and moved to a stronger channel binding.
4) Akamai servers disabled support or RSA-MD5 client and server signatures.
5) Red Hat issued backported patches RHEL 6 and 7 to disable MD5 signatures in their version of OpenSSL version 1.0.1e.
6) NSS 3.21 (FireFox 43) disabled support for MD5 server signatures; MD5-based client signatures were already disabled.
7) GnuTLS 3.3.15 disabled MD5 signatures in the default configuration.
8) BouncyCastle Java 1.54 (C# 1.8.1) disabled MD5 signatures in the default configuration.
9) Oracle and IBM are updating the TLS implementation in their Java runtimes to disable MD5 signatures in the default configuration.
10) mbedTLS is being updated to disable MD5 server signatures; MD5 client signatures were already disabled.

These changes impact the Firefox and Android browsers, about 31% of web servers, most Java application servers and their clients, and many other custom applications that use less well-known TLS libraries. We are maintaining a website with the currently known attacks, affected software, and disclosure status at our website:

http://sloth-attack.org .

VIII. Conclusions

We have demonstrated that the use of MD5 and truncated HMACs for authenticating transcripts in various Internet protocols leads to exploitable chosen-prefix and generic collision attacks. We also showed several unsafe uses of SHA-1 that will become dangerous when more efficient collision-finding algorithms for SHA-1 are discovered. In all cases, the complexity of our transcript collision attacks are significantly lower than the estimated work for a second preimage attack on the underlying hash function. This definitively settles the debate on whether the security of mainstream cryptographic protocols depend on collision resistance. The answer is *yes*, cryptographers were right. Except in rare cases, mainstream protocols do require collision resistance for protection against man-in-the-middle transcript collision attacks. Consequently, we strongly recommend that weak hash functions like MD5 and SHA-1 should not just be deprecated; they should be forcefully disabled in existing protocols.

An open research question is whether it is possible to design key exchange protocols that will be resilient to new collision attacks. One strategy is to use a commitment scheme (like ZRTP [40]) that would make it more difficult for a man-in-the-middle to tamper with the transcript. However, such schemes may still be vulnerable to certain shortcut collisions [19]. For signatures, randomized hashing [13] provides a different way forward but its integration into a complex protocol like TLS would need to be carefully analyzed.

Protocol	Property	Mechanism	Attack	Collision Type	Precomp.	Work/conn.	Preimage	Wall-clock time
TLS 1.2	Client Auth	RSA-MD5	Impersonation	Chosen Prefix		2^{39}	2^{128}	48 core hours
TLS 1.3	Server Auth	RSA-MD5	Impersonation	Chosen Prefix		2^{39}	2^{128}	48 core hours
TLS 1.0-1.2	Channel Binding	HMAC (96 bits)	Impersonation	Generic		2^{48}	2^{96}	80 GPU days
TLS 1.2	Server Auth	RSA-MD5	Impersonation	Generic	2^X conn.	2^{128-X}	2^{128}	
TLS 1.0-1.1	Handshake Integrity	MD5 | SHA-1	Downgrade	Chosen Prefix		2^{77}	2^{160}	
IKE v1	Initiator Auth	HMAC-MD5	Impersonation	Generic		2^{65}	2^{128}	
IKE v2	Initiator Auth	RSA-SHA-1	Impersonation	Chosen Prefix	2^{77}	0	2^{160}	
SSH-2	Exchange Integrity	SHA-1	Downgrade	Chosen Prefix		2^{77}	2^{160}	

TABLE I. SUMMARY OF TRANSCRIPT COLLISION ATTACKS ON INTERNET PROTOCOLS

REFERENCES

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, , and P. Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In *ACM CCS*, 2015.

[2] J. Altman, N. Williams, and L. Zhu. Channel bindings for TLS. IETF RFC 5929, 2010.

[3] S. Bellovin and E. Rescorla. Deploying a new hash algorithm. In *NDSS*, 2006.

[4] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila. Multi-ciphersuite security of the secure shell (ssh) protocol. In *ACM CCS*, pages 369–381, 2014.

[5] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti. Verified contributive channel bindings for compound authentication. In *NDSS*, 2015.

[6] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland)*, 2014.

[7] I. B. Damgård. A design principle for hash functions. In *CRYPTO'89*, 1990.

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[9] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2014.

[10] M. Friedl, N. Provos, and W. Simpson. Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol. IETF RFC 4419, 2006.

[11] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. In *ACM CCS*, 2013.

[12] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS. Internet Draft, 2015.

[13] S. Halevi and H. Krawczyk. Strengthening digital signatures via randomized hashing. In *CRYPTO*, 2006.

[14] B. Hill, D. Baghdasaryan, B. Blanke, R. Lindemann, and J. Hodges. FIDO UAF Application API and Transport Binding Specification v1.0. Draft Specification, 2015.

[15] P. Hoffman. Use of Hash Algorithms in Internet Key Exchange (IKE) and IPsec. IETF RFC 4894, 2007.

[16] P. Hoffman and B. Schneier. Attacks on Cryptographic Hashes in Internet Protocols. IETF RFC 4270, 2005.

[17] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, 2012.

[18] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, 2004.

[19] J. Kelsey and T. Kohno. Herding hash functions and the nostradamus attack. In *EUROCRYPT*, 2006.

[20] D. Knuth. Seminumerical algorithms, volume 2 of the art of computer programming, 1981.

[21] H. Krawczyk. SIGMA: The SIGn-and-MAc approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *CRYPTO*. 2003.

[22] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, 2013.

[23] A. Langley. Transport Layer Security (TLS) Snap Start. Internet Draft, 2010.

[24] A. Langley. Transport Layer Security (TLS) Next Protocol Negotiation Extension. Internet Draft, 2012.

[25] G. Leurent. Practical key-recovery attack against APOP, an MD5-based challenge-response authentication. *IJACT*, 1(1):32–46, 2008.

[26] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[27] F. Mendel, C. Rechberger, and M. Schläffer. MD5 is weaker than weak: Attacks on concatenated combiners. In *ASIACRYPT*, 2009.

[28] A. Menon-Sen, N. Williams, A. Melnikov, and C. Newman. Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms. IETF RFC 5802, 2010.

[29] R. C. Merkle. A certified digital signature. In *CRYPTO'89*, 1990.

[30] J. M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[31] J. M. Pollard. Monte carlo methods for index computation. *Mathematics of computation*, 32(143):918–924, 1978.

[32] A. Popov, M. Nystroem, D. Balfanz, and A. Langley. The Token Binding Protocol Version 1.0. Internet Draft, 2015.

[33] M. Stevens. Hashclash. https://marc-stevens.nl/p/hashclash/.

[34] M. Stevens. Counter-cryptanalysis. In *CRYPTO*, 2013.

[35] M. Stevens. New collision attacks on SHA-1 based on optimal joint local-collision analysis. In *EUROCRYPT*, 2013.

[36] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and applications. *IJACT*, 2(4):322–359, 2012.

[37] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[38] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT*, 2005.

[39] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), 2006.

[40] P. Zimmermann. ZRTP: Media Path Key Agreement for Unicast Secure RTP. IETF RFC 6189, 2012.

H	Collision	CPC
Generic	$2^{N/2}$	$2^{N/2}$
$H_1 \vert H_2$	$2^{N_1/2}N_2/2 + 2^{N_2/2}$	$2^{N_1/2}N_2/2 + 2^{N_2/2}$
MD5	2^{16}	2^{39}
SHA-1	2^{61}	2^{77}
MD5 | SHA-1	2^{67}	2^{77}

TABLE II. COMPLEXITY OF FINDING COLLISIONS IN VARIOUS HASH CONSTRUCTIONS

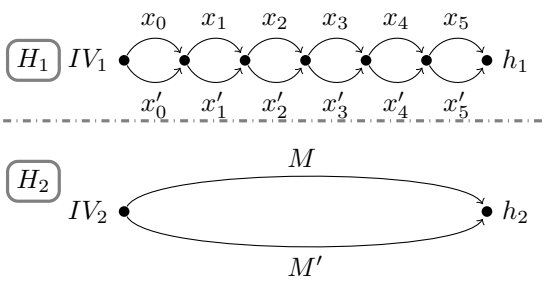

Fig. 9. Multi-collision attack

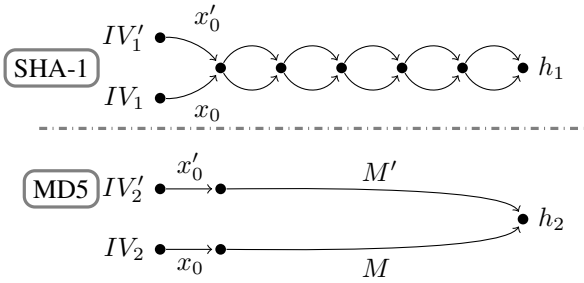

Fig. 10. CPC attack against MD5 | SHA-1

APPENDIX

A. Attacks against Hash Functions

We now give more details about attacks against hash function, considering both generic attacks and dedicated attacks against widely-used functions MD5 and SHA-1. The main results are summarizex in Table II.

Generic collision attacks While a basic collision attack requires to compute and store $2^{N/2}$ images of the hash function, it is possible to mount a parallel and memory-less attack with a very small overhead. The main idea was introduced by Pollard as the Rho algorithm for factorization [30] and discrete logarithms [31], and was later generalized to collision search. The hash function is first restricted from $\{0,1\}^* \to \{0,1\}^N$ to $\{0,1\}^N \to \{0,1\}^N$, so that it can be iterated. After some number of steps, a chain of iterations reaches a cycle, and the graph will have the shape of the greek letter ρ. On average, the cycle has length $O(2^{N/2})$ and is reached after $O(2^{N/2})$ steps. The point where the tail of the meets with the cycle reveals a collision in the hash function. It can be detected in time $O(2^{N/2})$ with little or no memory, using various cycle detection methods, such as Floyd's algorithm [20] (also known as tortoise and hare).

Some variants of this attack using distinguished points can be parallelized efficiently. We now describe a parallel version of Pollard's Lambda algorithm, as described by van Oorschot and Wiener [37], using c CPUs. Each CPU will compute iteration chains of the function H, and stop when reaching a *distinguished point*, that is a point with some easy to test property. For instance, we stop a computation when the ending point satisfies $x < 2^{N/2}\alpha c$ for some small constant α, so that the expected length of a chain is $2^{N/2}/\alpha c$. When a chain is finished, we store the starting point, the length, and the ending point. We generate αc chains in this way, so that the function has been evaluated about $2^{N/2}$ times, and there is a high probability that there was a collision. The important idea of this attack is that if a given point is reached by two different chains, both chain will stop at the same distinguished point. Therefore, we look at the ending points of the chains, and when a collision is detected, we restart the chains from the starting point in order to locate the collision. This attack requires about $2^{N/2}$ evaluations of H, and a memory of αc when using c CPUs.

This attack can be tweaked for a chosen-prefix collision attack using an auxiliary function $g\colon \{0,1\}^N \to \{0,1\}^N$ defined as:

$$g(x) = \begin{cases} H(P|x) & \text{if } x \text{ is even} \\ H(P'|x) & \text{if } x \text{ is odd.} \end{cases}$$

Collisions in g can be found with the previous techniques. With probability $1/2$ a collision is g is between an even x and an odd x' (or vice versa), this implies a chosen-prefix collision $H(P|x) = H(P'|x')$. An accurate complexity analysis is provided in [37]: a collision is expected to be found after $\sqrt{\pi 2^N/2}$ evaluations. For a chosen-prefix collision, we expect to find two collisions in g after $\sqrt{\pi 2^N}$ evaluations.

Concatenation Collisions in the concatenation of two hash functions $H_1|H_2$ can be found with roughly the same effort as breaking the strongest one of the two, using the multi-collision technique of Joux [18].

The adversary first finds a collision pair (x_0, x'_0) for H_1, starting from the initialization value of H_1. Then it finds a collision pair (x_1, x'_1) starting from $H_1(x_0) = H_1(x'_0)$. This defines 4 messages with the same H_1-digest: $x_0|x_1$, $x_0|x'_1$, $x'_0|x_1$, $x'_0|x'_1$. After $N_2/2$ steps, this defines a set of $2^{N_2/2}$ messages with the

same H_1-digest. With high probability, two of these messages have the same H_2-digest as well (see Figure 9). Therefore, one can find a collision in $H_1|H_2$ with a complexity only $N_2/2 \times 2^{N_1/2} + 2^{N_2/2}$. For MD5 | SHA-1, this translates to 2^{80}, roughly as much as a generic collision attack on SHA-1.

Better attacks against MD5 | SHA-1 result from the combination of Joux's multicollision technique with shortcut attacks against SHA-1. A collision attack can be build for a cost of $64 \times 2^{61} + 2^{64} \approx 2^{67}$ (building sequentially 64 collisions for MD5). For a chosen-prefix collision, we first perform a chosen-prefix collision against SHA-1, to generate messages (x, x') such that SHA-1$(P|x)$ = SHA-1$(P'|x')$. Then we build a multicollision in SHA-1 starting from this value, and we evaluate MD5 over a set of 2^{64} messages in order to find a collision. The total cost is about $2^{77} + 64 \times 2^{61} + 2^{64} \approx 2^{77}$ (see Figure 10).

Moreover it has been shown that it is possible to combine cryptanalytic shortcuts both on SHA-1 and MD5, assuming that collision attacks against SHA-1 improve in the future [27]. This may allow collision attacks against MD5 | SHA-1 with less than 2^{64} work. Table II summarizes the currently-known complexities for computing various hash collisions.

B. Transcript Collision Attacks on IKEv2 and SSH-2

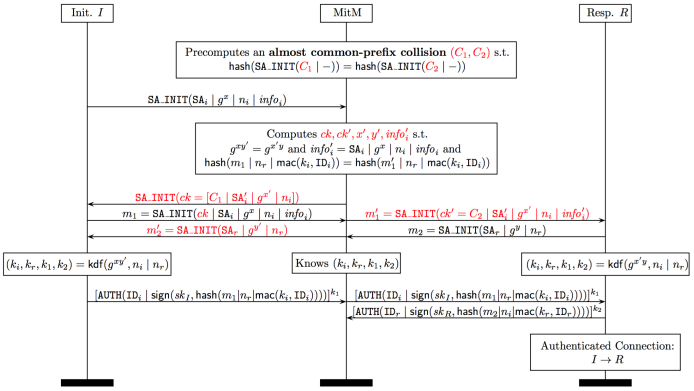

Fig. 11. Man-in-the-middle initiator impersonation attack on IKEv2. The initiator I connects to the responder R but a man-in-the-middle attacker A intercepts and tampers with some messages (shown in red). A pecomputes a collision (C_1, C_2) between the prefixes of two SA_INIT messages that both begin with a cookie payload. Then by sending a carefully crafted cookie to I, A can trigger a transcript collision on the initiator signature, which it can then forward to R, thereby impersonating I on a connection that it controls.

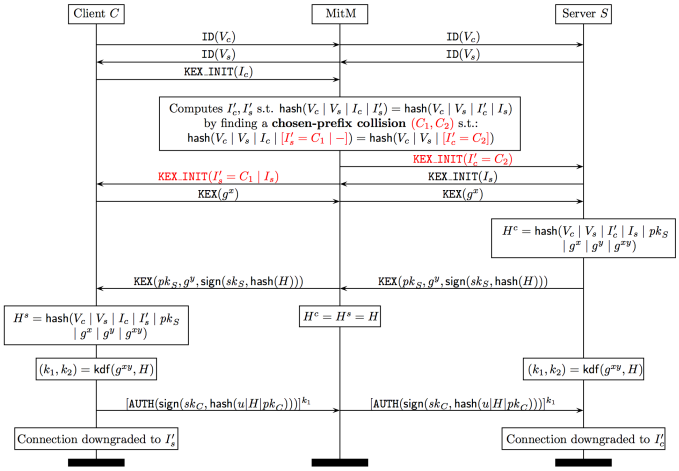

Fig. 12. Man-in-the-middle downgrade attack on SSH-2. The client C connects to a server S, but a network attacker A tampers with the key exchange messages (shown in red) to downgrade them to a weak ciphersuite. To succeed, A must compute a chosen-prefix collision on the session hash H after receiving C's key exchange message.