HASH CHAINS: A WEAK LINK FOR TRUSTED COMPUTING

JONATHAN A. PORITZ

ABSTRACT. *Trusted computing* is a phrase used, particularly by the member companies of the *Trusted Computing Group* (TCG), to describe an approach to IT security based on keeping a snapshot of a platform's execution history in a relatively secure location – for the TCG, this is a *Trusted Platform Module* (TPM) (roughly, a smartcard chip directly on the platform motherboard) – and then answering remote queries about the snapshot with signed data blobs. While there are many technically interesting questions which remain to be examined for the whole TCG scenario, we concentrate here on a basic technical consideration underlying this technology. In fact, we find a pair of issues that occur for any such security infrastructure that uses hash chaining as the fundamental forward-ratcheting device preventing malware from being installed but concealing its own traces.

The first of these might be called an "attack" on the basic security of the hash chaining mechanism ... were it not that this attack is presently infeasible. Nevertheless, recent advances in cryptanalysis of hash functions (up to and including the SHA1 hash function used by the TCG) might make the early adopters of a trusted computing scheme very nervous about the potential future catastrophe which would result from further progress against SHA1, particularly since the attack we propose leverages the work of a single cracker (perhaps supported by a grid of zombie machines doing collision searches) into a secret which can be used by an arbitrarily large number of others to break the security of their TPMs for many of the usecases the TCG encourages. We propose a fairly simple defense which renders this attack far more difficult – making the cracker search instead for a highly constrained hash collision. This fix can even be applied, albeit with some discomfort, by users of today's TPM, as we explain.

Our second problem is a severe one regarding the feasibility of application of trusted computing technologies due to scaling considerations. It is quite simple to state, being based merely upon the non-commutativity of hash chains, and also admits a simple solution where trusted computing is based instead on cryptographic accumulators. In fact, given the trust it is necessary to place in the TPM itself (or whatever similar mechanism replaces it in some other trusted computing proposal) and the trust it would be best to be able to place in the non-invertibility of the underlying hash function, we show how a much simpler cryptographic accumulator is possible than what is usually implied by that term.

In order to discuss conveniently issues surrounding hash chains and their security, we provide a new theoretical description of such cryptographic tools. This description is based on some concepts from abstract algebra and is of independent interest and application for cryptography.

1. BACKGROUND: TCG TECHNOLOGY

The *Trusted Computing Group* (TCG) is a consortium of electronics industry players whose stated *raison d'être* [10] is to develop and promote "open specifications" for a small piece of hardware, called the *trusted platform module* (TPM), which can be used to increase the security of today's computing platforms and, eventually, other of tomorrow's electronic devices – PDAs, cellphones, *etc.*Various of the TCG members have released products containing TPMs: in particular, it is estimated [6] by IT industry market analysts IDC that more than more

than 50 million TPMs will ship with laptop and desktop computers in 2006, more than 110 million in 2007, *etc.*; these are mostly in IBM ThinkPads and NetVistas, HP d530 desktops and various laptops, some Fujitsu laptops, and, in the near future, in various Dell laptops and desktops. Furthermore, Microsoft's *Next Generation Secure Computing Base* (NGSCB), a security strategy for future releases of their operating system, has many points of contact with particular features of the TPM, to the extent that NGSCB seems only possible in its fullest, more secure form on a platform which has a hardware TPM. A number of Linux kernel modules and user-space tools which work with TPMs are also being released by several corporate and open-source actors.

It therefore appears that the TCG grand strategy is being given a good trial in the world marketplace, at least in so far as widely distributing the necessary hardware and providing ample operating system support in OSes used by the great majority of users. One last component whose general availability is unclear at the moment is the infrastructure, public-key and other related cryptographic, which is necessary for the full TCG world-view. We back-track to describe TCG technology in greater detail so that we will be able to outline this infrastructure and place in context the issues of this paper.

1.1. **Tracking binaries executed.** As intimated above, trusted computing is built out of two fundamental pieces: an execution snapshot (or history) stored in the TPM and a set of protocols and supporting infrastructure elements to enable the platform to respond reliably to a remote site when it queries the current snapshot values. The TPM is attached to the main platform hardware in a tamper-evident way (such as, by being soldered to the motherboard). It has a small amount of its own memory (some of which is non-volatile) and computational power, and communicates with the host platform only via a limited, carefully-designed API. The non-volatile memory of the TPM can hold long-term secrets, such as the private part of an asymmetric key-pair which is generated on-chip and never leaves the TPM, and shorter-term data, such as the execution snapshot we have referred to several times.

The mechanism provided by the TCG for this last purpose is a collection of *platform configuration registers* (PCRs), which are reset at boot time and then updated every time a new executable is loaded, with the hash of that executable added to the end of a hash chain. If the BIOS and hardware can be trusted, then they form a root of trust for this hash chain: after the BIOS executes, the OS loader, the OS itself, and each new executable to be run within the OS are, in turn, hash chained into an appropriate PCR (these executables are *measured*, in the parlance of the TCG), each hash chaining occurring *before* execution control is passed to the new binary. In this way, while certainly binaries may be loaded which seize control of the OS and refuse to continue the TCG contract of measuring before executing further binaries, these negligent binaries will have themselves been measured and so the PCR values will have the tell-tale traces of some faulty binary already registered.

As a particular example, if some host has BIOS which hashes to the value β, OS loader to λ, OS kernel to κ, loaded OS modules (such as drivers for specialized hardware) with hashes $\chi_1, \ldots, \chi_{n_m}$, loaded user-space applications with hashes $\chi_{n_m+1}, \ldots, \chi_{n_m+n_a}$, then some PCR will have value

$$(1.1) \qquad PCR_j = H(H(\ldots H(H(H(H(0^{160} \parallel \beta) \parallel \lambda) \parallel \kappa) \parallel \chi_1) \parallel \ldots) \parallel \chi_{n_m+n_a})$$

where here $H(\cdot)$ is the hash function (SHA1 for the TCG), 0^{160} is a 160-bit string of 0s, and $\parallel$ represents concatenation. Another, perhaps easier, way to think of this is as initializing the PCR by

$$(1.2) \qquad\qquad\qquad\qquad\qquad PCR_j = 0^{160}$$

and then repeatedly updating this PCR ("extending the hash chain") with

$$(1.3) \qquad\qquad PCR_j := H(PCR_j \parallel x)$$

as x ranges sequentially through the values β, λ, κ, and then $\chi_1, \ldots, \chi_{n_m+n_a}$.

[Note this is a small simplification of the actual TCG procedure, which allows for finer-grained measurements in different PCRs, sometimes initializing the PCR with some value other than 0^{160} – which value is, however, fixed – and other minor changes which do not detract from the essential points we are making in this paper.]

1.2. **Attestation.** The second basic feature of a trusted computing solution is the ability of the platform to report reliably its current execution snapshot – which we now see as encoded in PCR values – to a remote site. For our purposes here, the precise mechanism does not matter, it suffices to know that a remote site can request a *TPM Quote*, which is a data blob containing the current values of (some of) the PCRs and followed by a digital signature. This signature is built using an identity the host has previously constructed, in negotiations with a trusted third party and using various vendor certificates and other data. The entire protocol for creating and conveying this quote is called *remote attestation*, and the particular platform is said to "attest to its security state".

We do not at all presently call into question these protocols, as the issues we are about to investigate remain even if we assume this heavy-weight part of the TCG infrastructure is entirely without flaw. One small feature that will be relevant, however, is worth mentioning: as part of the remote attestation protocol, the platform sends a log file containing information about the binaries in the execution history, including some identifier of the binary and its hash, *as well as* the TPM Quote. The remote verifier can trust the Quote, because it is signed with keys coming from the whole PKI of platform/user/pseudonym identities, and the log file can then be trusted after the verifier reconstructs the hash chain itself and checks that it equals the Quoted value.

1.3. **A use case.** Let us complete our description of this TCG background by describing how their approach could be used, were it to be entirely secure, to enable certain networked electronic commerce. As an example, we consider the application to *digital rights management* (DRM) – while it is claimed in the TCG promotional literature that DRM was not a particular goal of the TCG, certainly their technology can easily be used for quite sophisticated DRM applications. If the mention of DRM makes the reader uncomfortable, the scenario we describe also applies almost without change to a situation in which an e-banking client is running on a TPM-enabled host, or any other in which a server wishes to reveal secrets only to clients who can prove their security state with TPM-assisted mechanisms.

So imagine a situation in which a platform user wishes access to valuable new digital content, available from a content-provider's web site ... should the provider have the confidence that the client will not be copying that data then to post it to a file-sharing network. The content provider may have a policy whereby a certain vendor's hardware, running a particular vendor's OS and a particular media player, is considered safe. TCG identity certificate attributes (and possibly some information in certain PCRs) could be used to show the hardware was acceptable, while the recognized OS and player – and no cracker's eavesdropping software! – could be known by the provider to be running on the client by examining the output of a TPM Quote. The OS and media player could even demonstrate, via these mechanisms, that they are in a state whereby no other, potentially content-stealing, application would possibly be initiated, until the content stream was stopped by the provider.

This is the fundamental use case we shall consider, and it makes clear what the "remote verifier" (the content provider, in this case) needs to see and needs to trust in order to have confidence in the user's platform.

2. Digression to abstraction

It will be much easier to work with hash chains if we set them in a somewhat more abstract setting. Stepping back a moment, then, we view a hash chain, formed as described above in §1, as the value of single register (a PCR, for the TCG), which contains the outputs of the hash function. So let $\mathcal{R}$ (for *register* or *range*) denote the set of well-formed hash function outputs; for the TCG, $\mathcal{R} = \{0,1\}^{160}$ (and in most other reasonable situations it will be $\{0,1\}^k$ for some k). Then notice that the process of using our hash function $\mathcal{H} : \{0,1\}^* \to \mathcal{R}$ for a hash chain essentially uses $\mathcal{H}$ to endow $\mathcal{R}$ with a binary operation, which we shall write multiplicatively with a "$\cdot$":

$$(2.1) \qquad \mathcal{R} \times \mathcal{R} \to \mathcal{R} : (\alpha, \beta) \mapsto \alpha \cdot \beta = \mathcal{H}(\alpha \parallel \beta)$$

Note, however, that this binary operation has very few properties: it is not likely to be Abelian or even associative, so in the terminology of abstract algebra, it makes $\mathcal{R}$ into a *magma* (or sometimes this is called a *groupoid*, however the latter term is more commonly used with an entirely different meaning, so we avoid it here).

2.1. **Conventions.** With this notation and operation understood, we would like to write (1.1) as

$$(2.2) \qquad PCR_j = 0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \chi_1 \cdot \cdots \cdot \chi_{n_m + n_a}$$

although, strictly speaking, this is ambiguous, as $\mathcal{R}$ with the magma structure induced by $\mathcal{H}$ and (2.1) is not necessarily (in fact, not likely) associative. It does make sense, however, to initialize PCR_j with (1.2) and then to define inductively, in perfect analogy to (1.3)

$$(2.3) \qquad PCR_j := PCR_j \cdot x$$

as x ranges sequentially through the values β, λ, κ, and then $\chi_1, \ldots, \chi_{n_m+n_a}$.

We shall in fact henceforth adopt the notation implied by this inductive definition, not just for the PCR_j just mentioned, but for all products of several terms: that is, whenever we write a product of several terms, using the binary operation of the magma, we shall mean that the product is evaluated left to right. Hence, under this notational convention, the definition of PCR_j in (2.2) does make sense, and coincides with the definition given by (1.2) and (1.3).

Of course, our magma abstraction is useful mainly in writing expressions and suggesting traditional algebraic manipulations, but it conceals the actual application of this technology in some ways, one of which will become useful later in this paper. The essence of what is excessively concealed is that expressions like (2.2) (especially its right-hand side) really pertain to the value *in a particular PCR* (although the left-hand side makes reference to this fact) in the TPM *at a particular moment in time – i.e.*, the PCRs are *stateful*; we will use the statefulness in particular, below.

2.2. **Quasigroups.** We finish this digression with a few words about division. For a fixed $\alpha \in \mathcal{R}$, we would expect that the left multiplication by α map, $L_\alpha : \mathcal{R} \to \mathcal{R} : x \mapsto \alpha \cdot x$, would be a permutation of $\mathcal{R}$ – if not, $\mathcal{H}$ is surely a rather poor hash function. It follows that for any $\omega \in \mathcal{R}$, there should exist some $x \in \mathcal{R}$ such that $\alpha \cdot x = \omega$; this x is usually denoted $\alpha \backslash \omega$. Similar thoughts about right multiplication lead to our believing in the existence of a $y = \omega / \alpha$ with the property that $y \cdot \alpha = \omega$. Hence $(\mathcal{R}, \cdot)$ is a *quasigroup* and not merely a magma.

It is possible to study these magmas/quasigroups coming from hash chains using the technology of theoretical cryptography. We do not choose to do so here, since the point of this paper is to examine some very practical issues for the TCG.

3. A "SPRING BREAK ATTACK"

We continue with the DRM scenario as above in §1, and now describe the attack. Imagine an enterprising but unethical college student wishes to steal digital content. She presumably has access to software modules which can be run in user-space on her platform which will take sufficient control of the OS (a) to place the measuring and hash chaining functionality on her platform completely in her hands, and (b) to install spying software which will steal the digital content stream, should the content provider be so foolish as to serve it to her. We will imagine these two cracks are realized by two separate binaries, the first of which has hash value c (the hash value of the second cracking tool's binary can remain unspecified), and we imagine also that they are fairly easily available to our would-be cracker – this is consistent with the rate of flaws discovered in market-dominant operating systems, especially when we consider that the content provider would likely accept even fairly slowly updated OSes, OS modules (drivers), and media players as still valid, since to do otherwise would be to exclude the large majority of the potential customer market which does not instantly upgrade to bleeding-edge hardware and software whenever possible.

3.1. **Naïve cracking approach.** Our cracker has a problem: the hash chain in some PCR will be wrong after she runs her cracktools. The content provider is willing to serve the digital content to platforms running (as before) BIOS β, OS loader λ, OS kernel κ, OS modules $\chi_1, \ldots, \chi_{n_m}$ and applications $\chi_{n_m+1}, \ldots, \chi_{n_m+n_a}$ – one of which applications is presumably the recognized media player. Therefore the content provider is expecting the appropriate PCR to have value

$$(3.1) \qquad h_{\text{ok}} = 0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \chi_{\rho(1)} \cdot \cdots \cdot \chi_{\rho(v)}$$

where ρ is a permutation of the integers $\{1, \ldots, v\}$ for some $v \leq n_a + n_m$; if the cracker actually loaded only the modules and applications with hashes $\mathcal{N} = \{\nu_1, \ldots, \nu_w\} \subseteq \{\chi_1, \ldots, \chi_{n_m+n_a}\}$ (where $w \leq n_m + n_a$) and then the cracktool c, the actual PCR value would be:

$$pcr_{\text{actual}} = h_{\text{base}} \cdot c$$

where

$$h_{\text{base}} = 0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \nu_1 \cdot \cdots \cdot \nu_w.$$

But our cracker has control of the hash chaining now! So if only she could find a quasigroup quotient with denominator c, she could chain in that value to the appropriate PCR, whereafter her TPM would report to the content provider, in exactly the usual, "trustworthy" manner, a configuration corresponding to PCR value determined by the numerator of that quotient, even thought the platform would in fact be far from having that configuration. If the numerator can be correctly chosen, then, the cracker can simulate a trustworthy platform, meanwhile instead installing the media-stealing software, without updating the hash chain ... hence completely vitiating the TPM enforcement of digital rights. Note also that the quotient of which we speak is a single value out of $\mathcal{R}$, so merely a bit string of length 160; this could be published on a web site and then anyone who boots a platform which has a similar h_{base} would also be able to use this quotient and the same cracking tools to seize control of their platform away from the TPM – *i.e.*, this TPM-cracking secret is generically useful for all would-be content pirates, it

does not depend on any features of the platform itself, or particular platform or user or owner credentials or certificates.

3.2. Specific attack.

Let us make explicit the computational problem which the cracker must solve: the set $\{\beta, \lambda, \kappa, \chi_1, \ldots \chi_{n_m+n_a}\}$ is likely to remain constant (or, at least, only grow to include more kernel modules and applications) for months or longer. The cracker seeks any value in $\mathcal{R}$ (a 160-bit value)

$$(3.2) \qquad\qquad q = (h_{\text{base}} \cdot c)\backslash h_{\text{ok}},$$

so that

$$(3.3) \qquad\qquad pcr_{\text{actual}} \cdot q = h_{\text{ok}},$$

(for some permutation ρ) since this last is an acceptable PCR value, in the eyes of the content provider.

We imagine our cracker has the personal discipline to skip spring break in order to hijack all of the idle computers on her college campus during that vacation, during which time she runs a distributed, brute-force (but brute-force informed by the latest known weaknesses in SHA1!) search for one of the magic values q. When she finds this value, she posts $\{\beta, \lambda, \kappa, \chi_{\rho(1)}, \ldots \chi_{\rho(v)}, q\}$ on her web site, and from then on, anyone using the same OS, modules, applications and cracking software c can steal digital content (and interfere with e-banking transactions and overthrow all secret empires based on TCG, *etc.*, *etc.*).

4. (Im)Practicalities of this attack and (non)trivialities of a fix

4.1. Some big numbers.

A state-of-the-art SHA1 implementation can steam about 116MB/sec on a typical consumer PC running at 2.6GHz. This translates to testing about 2^{21} possible values for a particular quotient q per second on a single PC, and perhaps 2^{30} per second on a zombie network. If SHA1 were a perfect hash function, our cracker would have to try 2^{160} possible values, which amounts to 2^{130} seconds. As the age of the universe is currently estimated to be about 2^{58} seconds, this is unlikely to be a successful attack in any given spring break.

Despite these impressive numbers, it is probably unwise to have too much confidence in the security of digital content protected by this mechanism, at least for anything other than the immediate future. A whole series of recent advances in the cryptanalysis of SHA1 (see [15], [12], [4], [5], [7], [8]) lead in the general direction of drastically reducing the searching required to find collisions. In particular, the note [13] (without details, as of the time of this writing, but from a reliable team), claims a reduction of the number of runs of SHA1 needed to find a collision from the expected 2^{80} ($= \sqrt{2^{160}}$) down to 2^{69} – not breakable by today's technology, but suggestive. Even more recent work by the same authors (see [14], but the Crypto '05 Rump session presentation was even more ambitious) points towards complexities of perhaps 2^{63}, which *is* breakable today (or, at least, some computations of similar complexity have been performed, albeit after a very large investment).

It is also worth noticing that published attacks on hash functions (due to Joux, [5] and [4]; explained simply in [8]) are based upon its block structure. But the hash chain itself (2.2) is a further layering of block structure on top of the bare hash function, and these blocks at a higher level may be permuted as the cracker desires, so there is a high chance that some permutation of the, say, $n_m + n_a = 30$ kernel modules and user applications is likely to have a particular form useful for collision attacks. In fact, the cracker, having seized control of the OS, can extend the PCRs by *arbitrary* further bit strings of length 160, so the cracker's task is to find ordered

subsets $\Phi = \{\varphi_1, \ldots, \varphi_r\}$ and $\mathcal{R}$ and $\mathcal{S} = \{j_1, \ldots, j_s\} \subseteq \{1, \ldots, n_m + n_a\}$ and to compute the quotient

$$(4.1) \qquad q_{\Phi, \mathcal{S}} = (h_{\text{base}} \cdot c \cdot \varphi_1 \cdot \cdots \cdot \varphi_r) \backslash \left(0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \chi_{j_1} \cdot \cdots \cdot \chi_{j_s}\right)$$

which is more in the nature of collision search than search for a particular hash function inverse.

Nevertheless, as of the time of this writing, the numbers are clearly not such as to allow a simple computation of $q_{\Phi, \mathcal{S}}$. We are then in a strange twilight where the hash chain underlying all of the TCG's security is not yet so weak as in fact to undermine all of the TCG's security ... but this abysmal future could well be just over the horizon. The obvious response to this is to rewrite all the TCG specifications to use a better hash function (or to permit flexibility in the choice of hash function), and the TCG is in the process of doing exactly this, but SHA1 is so fundamentally wired into these specifications it is not an easy or quick task. Furthermore, if some new security infrastructure is to be built that might last for as long as possible, we feel it is appropriate to consider designs which will continue to guarantee as much security as is possible, even while the underlying hash function is found to be weaker and weaker.

4.2. **Fundamentally....counting.** Let us then consider the basic issue being exploited in the attack we have described, and the compromises which must be taken into consideration in designing a response. The basic issue is, as we have seen, that the only thing proving the validity of the execution history log file which is transmitted by the (potentially compromised) OS to a remote verifier is the final value of the hash value, as sent to the verifier in the signed blob from a TPM Quote. After all, we are assuming that a cracker can seize complete control of her platform – that is, the normal part of the platform, not the inside of the TPM itself – so the communication between the platform's TPM and the remote verifier, and the entire communication of the log file, is passing through the hands of the cracker. As the TPM Quote is signed with a key that is in the TPM and not controlled by the cracker, the PCR value reported in a (signed) Quote is indeed the actual value on the TPM, no cracker modifications can take place (assuming, of course, the security of all keys and the strength of the underlying public key cryptography).

Therefore any attack (in the security model where the TPM remains inviolate) must be based upon breaking the association between a particular 160-bit value and an entire sequence of binaries which have been executed. It is reasonable to assume that in the near future SHA1 may become less resistant to the collision search-type attacks of (4.1), while perhaps complete inversion of SHA1 may be longer in coming (if every).

Thus we imagine that the particular mapping from binaries to 160-bit bit strings can be trusted, but it is the chain itself that is the weak point. A first step in compensating for this potential attack would be, upon examining the simpler (3.2), to keep track of *how many binaries were chained* to form the current value of the PCR. This would prevent the successful application of (3.2) and (3.3), since while the PCR value would be correct, the count of binaries chained would be two greater than that expected by the verifier, as c and then its cancelling quotient q would have been extended in the hash chain, although not in a way visible to the verifier by looking only at the PCR value.

From the abstract algebra point of view of §2, we are suggesting that each PCR also keep track of the *word length* of the expression whose value it contains. Of course, this is not, strictly speaking, well defined, but it becomes so with respect to a set of generators – the hashes of the binaries one expects to see on a typical platform – which set is quite small relative to the size of set $\mathcal{R}$ of hash output values. We would expect, due to similar reasoning as that in

§2, that the algebraic operation of the magma structure on $\mathcal{R}$ would be unlikely to have any word-shortening relations for any particular (relatively small) base.

Being more precise about this word length counter defense, with the notation of the more complex (and more likely successful) attack in (4.1): the cracker can, if she can compute $q_{\Phi,\mathcal{S}}$, set the PCR to a recognized value of

$$0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \chi_{j_1} \cdot, \cdots \cdot \chi_{j_s}$$

with word length $3 + s$, after in reality forming the chain

$$0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \nu_1 \cdot \cdots \cdot \nu_w \cdot c \cdot \varphi_1 \cdot \cdots \cdot \varphi_r \cdot q_{\Phi,\mathcal{S}}$$

which used $3 + w + 1 + r + 1$ extensions. Hence, if the TPM reports in each signed Quote the number of extensions represented by the values of the PCRs reported, then the verifier will recognize that the PCR values have been modified unless

(4.2) $$s = w + r + 2.$$

This simple constraint may or may not be enough to render the collision search impracticable.

4.3. **Counting on [in] TPMs.** As mentioned above, trusted platform security must rest on the PCRs in the TPM and, in particular, in the *signed* blob produced in a TPM Quote, as the operating system is presumed to be putty in the hands of the cracker so that separate side channels are completely untrustworthy unless they are, as the TCG model supposes, protected by cross-referencing with the contents of the signed Quote (although we have seen that this confidence is perhaps somewhat misplaced). Similar reasoning shows that the word length count we are proposing as a defense *must be delivered in a signed communication* from the TPM to the verifier. Thus this defense requires some modification of the TPM specification, either to add certain fields to the output of the Quote, or to add commands which return this information (directly, or hooked to certain Quoted PCR values). The additions to the specification are quite simple to design: new monotonic counters (a concept which already exists in the spec) would be added, one per PCR. Language would be added to indicate that these counters are set to 0 on power-on and reset to 0 only when a particular PCR is itself reset (the possibility to reset PCRs being a attribute of selected such registers). Finally, a command `TPM_CountedQuote` would be added which looks exactly like `TPM_Quote`, but which includes also the PCR count value for each PCR reported.

Note that the fix above is only effective if the constraint (4.2) pushes the amount of work to be done in finding the quotient $q_{\Phi,\mathcal{S}}$ beyond reasonableness (even by a cracker with a long spring break and a large zombie net). Therefore, whether or not it is worth rushing out an emergency revision of the TPM specification with this fix is unclear, but the situation will clarify as more results are published in the next months and even year or two about cryptanalysis of SHA1. In any case, given how easy and simple the fix is, future regular specification releases certainly should, in our view, include this feature to allow at least the simple-minded defense against the same type of attack against even the improved hash chains of the future (such as simply replacing SHA1 with SHA-256).

In fact, the current release of the TCG specification for the TPM has a feature which is, in certain ways, stronger than the simple-minded solution proposed above – although it is also weaker in other ways. The feature is called *auditing* by the TCG, and works essentially as follows. When the TPM is turned on, it resets an additional 160-bit hash chain register, beyond all the PCRs, called the `auditDigest`, as well as a counter called the `auditMonotonicCounter`. The TPM Owner can, using her Owner authentication, turn on the auditing function for certain TPM commands. When such a command $\mathcal{C}$ is then to be executed, the TPM first extends

the `auditDigest` by the concatenation of $\mathcal{C}$'s input parameters, $\mathcal{C}$'s op-code and the value of ++`auditMonotonicCounter`. After the successful completion of the command, `auditDigest` is extended again, by the concatenation of the output parameters of $\mathcal{C}$, its return code, its op-code, and the value of `auditMonotonicCounter`. A remote verifier could then require that `TPM_Extend` (the basic command which extends a PCR's hash chain) always be an audited command and then require every TPM Quote also be accompanied by the output of a `TPM_GetAuditDigestSigned`. The verifier would then have a trustworthy source of information source about the number of extensions used to construct a particular PCR.

This solution uses the current TPM specification and hence does not require any hardware modification – *e.g.*, of the millions of TPMs already in users' platforms. However, the TPM audit process consumes and generates far too much data, well beyond what our above simple-minded attack prevention required. In fact, the TPM specification [11] states, of the `auditDigest`, that

> "This value may be unique to an individual TPM. The value however will be changing at a rate set by the TPM Owner. Those attempting to use this value may find it changing without their knowledge. This value represents a very poor source of tracking uniqueness."

In addition, this tracking mechanism was clearly not designed with the use we propose in mind, hence it has an insufficiently fine-grained invocation: the audit process may be turned on for a particular *TPM command*, such as `TPM_Extend`, but *not for* `TPM_Extend` *when it acts on particular PCRs*, and *there are not one* `auditDigest` *and one* `auditMonotonicCounter` *available per each PCR whose word length we would like to count.*

As a consequence, using auditing to implement our suggested defense is not a recommended course of action. Nevertheless, because of the installed base of revision 1.2 (and even revision 1.1b!) TPMs, it is possible that a verifier will have to fall back to using auditing in the way we just described, when the client is an old TPM and if, in the near future, SHA1 falls further.

4.4. **Ordered counting.** In fact, a closer examination of the TPM auditing function shows that it is really implementing a stronger defense than the one we suggest, albeit decorated by too much other data, as we have noted. In order to be more precise about this, we shall continue to use the magma-style notation of §2, but we shall also keep in mind the statefulness of the TPM. So in fact an expression like that of (2.2) saying

$$(4.3) \qquad PCR_j = 0^{160} \cdot \beta \cdot \lambda \cdot \kappa \cdot \ldots$$

which *defines a PCR from its very initial value* could in fact be thought of as an initialization

$$(4.4) \qquad PCR_j = 0^{160}$$

that implicitly also initializes other state information, followed by the extensions

$$(4.5) \qquad PCR_j := PCR_j \bullet \beta, \quad PCR_j := PCR_j \bullet \lambda, \quad \ldots, \quad PCR_j := PCR_j \bullet \chi_{n_m+n_a}$$

where the new extension/magma operation is defined, using the current state information that this is the ℓth operation for PCR_j, as

$$(4.6) \qquad \alpha \bullet \beta = \mathfrak{H}(\alpha, \beta, j, \ell)$$

for $\mathfrak{H} : \mathcal{R} \times \mathcal{R} \times \mathbb{N} \times \mathbb{N} \to \mathcal{R}$ some decorated (keyed!) function based on hashing.

As an example of this new notation, consider that what is useful in the TPM Audit approach to blocking the spring break attack of §3: if the stateful magma operation is defined as in (4.6) with

$$(4.7) \qquad \mathfrak{H}(\alpha, \beta, j, \ell) = \mathcal{H}(\alpha \parallel \beta \parallel (\ell \mod 2^{160})),$$

wherein ℓ is written out in binary and then truncated at 160 bits, then not only is the word length represented by the PCR value apparent to a remote verifier, but so also is the *order* in which the hashes of the binaries were chained into the particular PCR. This issue of ordering is the central one of §5, below, but notice that the ordering constraint is a much tighter one than the simple one of (4.2), so hence may be useful in boosting the difficulty of the search for a quotient for the spring break attack beyond the resources of the potential cracker.

In fact, continuing in this vein, (4.7) could be replaced by something such as

$$(4.8) \qquad \mathfrak{H}(\alpha, \beta, j, \ell) = \mathcal{H}(\alpha \parallel \beta \parallel \Psi_{j,\text{etc.}}(\ell))$$

for some pseudo-random sequence $\Psi_{j,\text{etc.}}(\ell)$, keyed by the PCR number j and features ("etc.") of the TPM identity and perhaps the particular verifier and other localization data, with values in $\mathcal{R}$. There is no obvious way (and certainly no clean way) to do this with the 1.2 version TPM specification, since the crucial point here is that the operation of (4.8) is best performed *in the TPM*. However, if a formula like this is used in a future specification revision, then our hypothetical cracker of §3 could conceivably find a quotient which enables the imagined attack – but only for one particular instance of the pseudorandom sequence. Hence, this data could not then be published and used by every other potential thief of digital content. (Although for the particular use case of DRM, a single cracker stealing the content seriously reduces the content owner's revenue stream, as the content need only be stolen once and then posted on a peer-to-peer network.) Further related defensive measures are considered in the next section.

5. Accumulating Trust

In this section we consider an issue somewhat different from, but related to, the attacks and defenses described in §§3 and 4. So doing, we find a usability problem with hash chain-based trusted computing which will be an immediate problem for the TCG's first imagined applications. We present a first, simple solution to this problem which, however, requires reworking of the TPM specification, hence is not likely to find immediate approval in industry.

The problem we find in the revision 1.2 TPM was already mentioned in §4.4, where it was in fact given as an advantage in fighting §3's attack: the operation of the TCG's hash chain is *non-commutative*. This adds constraints which can increase the computational difficulty of our cracker's attack, and also limits the applicability of the result – but it adds a difficulty also for legitimate users of TCG technology. The point here is that remote verifiers are expecting, as was explained in §3, that legitimate users will have relevant PCRs containing values of the form of the h_{ok} in (3.1), *for some value $v \leq n_a + n_m$ and any permutation ρ of the integers $\{1, \ldots, v\}$* – but there are more than $(n_a + n_m)!$ such permutations, and as a typical machine may easily have many dozens or even a few hundred "legitimate" kernel modules and user applications, this number of legitimate values h_{ok} is again of the order of the big numbers of §4.1.

This is a concern for the TCG immediately, *i.e.*, it does not await further improvements in the cryptanalysis of SHA1. It precludes many of the simplest use cases and remote attestation protocols. For example, the factorial growth of the number of acceptable hash-chain values h_{ok} means that no remote verifier can store these acceptable values and simply check a Quoted value against a table – even if $n_a + n_m$ is fairly small, its factorial can be unmanageably large. Hence every remote verification *must* include the side-channel transmission of the execution history log and the verifier must recompute the corresponding hash value on the fly. Especially for centralized servers, this represents a potentially enormous computational and communications overhead.

What, then, can be done? Some subset of the permitted $n_a + n_m$ binaries will be chained into the relevant PCR, and the remote verifier wishes merely to know, in some sense, one

bit of output: does the PCR value represent a final value in a chain coming from a subset of these permitted binaries, or has another, unacceptable binary also been executed? One sensible approach is to have the PCR value looked up against acceptable values by a trusted third party (TTP) which is closer to the client machine, knows its typical usage histories and can verify its PCR values against (perhaps the factorial of) a much smaller number of usual acceptable binaries, then this TTP can pass on the summary information to the original remote verifier; this is the approach (in the broadest sense) of "property-based attestation", proposed, for example, in [9]. But this only somewhat mitigates the problem by pushing it closer to the clients, where $n_a + n_m$ may be expected to be smaller, at the cost of a far more complicated trust relationship, and all of the various overhead of a TTP.

Another solution, which would, however, require rewriting the TPM spec, would be to have (some) PCRs where the extension operation was *commutative.* In this way the list of executables loaded would have to be transmitted to the remote verifier, but not its order. In fact, if a canonical ordering of the executable names/labels could be decided upon (and perhaps even if there were a widely accepted canonical total list of acceptable executables), this list giving the platform's execution history could be transmitted in an extremely compact form (as a bit string from the total list of acceptable binaries, or as the hash of the canonically ordered list). In fact, in many situations where there were fairly small values of $n_a + n_m$, it would make sense not to transmit the execution history at all, and for the remote verifier simply to keep a list of *all* acceptable hash values and to trust the client platform if and only if its Quoted PCRs were such values; after all, there are merely $2^{n_a+n_m}$ such, which could well be small enough to manage.

Commutative analogues to the hash chain are known as *cryptographic accumulators* and have been studied in some depth, see, *e.g.*, [2], [1], [3], *etc..* The difficulty with using these constructions in our current context is that they are somewhat heavy in public-key cryptographic content, which is unfortunate given that the TPM is so computationally weak – although properties of these accumulators are more rigorously established and they admit certain nice extra features, such as zero knowledge proofs of knowledge of various properties. However, we give one extremely simple accumulator-like device which uses only the black art of hash functions.

So let us assume that the TCG moves in the near future to using the hash function SHA-256, which we shall write $H : \{0,1\}^* \to \mathcal{R} = \{0,1\}^{256}$, and let us also assume that inverting or finding collisions in this function is entirely impractical for the foreseeable future. Then the easiest accumulating PCR would be to use initialization

$$(5.1) \qquad\qquad\qquad PCR_j = 0^{256}$$

(like (1.2) but with more 0's) and then inductive step as in (1.3)) but with

$$(5.2) \qquad\qquad\qquad \alpha \cdot \beta = H(\alpha) + H(\beta) \mod 2^{256}.$$

This is clearly Abelian and has all of the advantages (of slower growth of number of acceptable accumulator values, *etc.*). Its main drawback is greater dependence upon the non-invertibility of the hash function, due to an obvious attack: if our cracker from §3 can compute the value

$$(5.3) \qquad\qquad\qquad \delta = H^{-1}(H(\chi) - H(c)) \in \mathcal{R},$$

for some χ equalling the hash of an acceptable binary, then the TCG security again comes crashing down – the last accumulated value appears to the remote verifier to be χ, not the cracktool. We note in passing, however, that computing the inverse in (5.3) is the full inversion problem, not an easier collision search. Note, also, that various keyed hashes could be used, in the flavor of (4.4), (4.5), (4.6) and (4.7) or (4.8), in order to constrain further the inversion

problem (although the gains are not large), although care must be taken not to remove entirely the advantage of the commutativity by forcing the keyed accumulator value to be recalculated each time by the server during the process of remote attestation.

6. Conclusions

We have examined some issues for the project of "trusted computing", in the particular form proposed by the Trusted Computing Group, around its fundamental security feature of a *hash chain*. One such is a serious weakness in the TCG's basic efficacy, in the light of today's ever increasing concerns about the strength of the hash function SHA1. We described in some detail the attack, enough to show it to be more in the nature of a collision search than an actual inversion of the hash function (which makes it all the more terrifying), and using a convenient new notation for hash chain mechanisms. We also found a way to strengthen somewhat the security of the TCG when its specification uses a hash function which is not reliably collision-resistant; this defense requires some reworking of (addition to) the TPM specification, or else, much more inconveniently, builds upon (the misuse of) an existing TPM feature.

Another issue was also found which is of immediate and large consequence for the usability of TCG technology, coming from a *factorial* scaling of acceptable hash chain values which a verifier must recognize. A solution is also proposed to this which, however, can *only* be implemented by a reworking of the specification. A sample such reworking is described, which avoids the more cryptographically sophisticated – but computationally expensive! – approach of cryptographic accumulators, although this simple sample relies very heavily upon the non-invertibility of a hash function.

All told, we have found several points which should be taken into consideration in designing future trusted computing solutions, or revising existing ones – including an attack against the TCG's trusted computing methodology which may become a serious problem in the near future – and we have introduced a notation and point of view on these mechanisms which should have continuing wide applicability.

References

1. N. Barić and B. Pfitzmann, *Collision-free accumulators and fail-stop signature schemes without trees*, EURO-CRYPT '97: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (London, UK), LNCS, vol. 1233, Springer-Verlag, 1997, pp. 480–494.
2. J. Benaloh and M. de Mare, *One-way accumulators: a decentralized alternative to digital signatures*, EURO-CRYPT '93: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (London, UK), LNCS, vol. 765, Springer-Verlag, 1993, pp. 274–285.
3. J. Camensich and A. Lysyanskaya, *Dynamic accumulators and applications to efficient revocation of anonymous credentials*, CRYPT '02: Advances in Cryptology (London, UK), LNCS, vol. 2442, Springer-Verlag, 2002, pp. 61–76.
4. A. Joux, *Collisions for SHA0*, presented at the Crypto 2004 rump session, 2004.
5. _____, *Multicollisions in iterated hash functions. applications to cascaded constructions*, CRYPTO '04: Proceedings of the 24th Annual International Cryptology Conference on Advances in Cryptology (New York), LNCS, vol. 3152, Springer-Verlag, 2004, pp. 306–316.
6. R. Kay, *The future of trusted computing*, Commissioned Research Report, on TCG web site, 2005, `https://www.trustedcomputinggroup.org/home/IDC_Presentation.pdf`.
7. J. Kelsey and B. Schneier, *Second preimages on n-bit hash functions for much less than 2^n work*, Cryptology ePrint Archive, Report 2004/304, 2004, `http://eprint.iacr.org/`.
8. A. Lenstra, *Further progress in hashing cryptanalysis*, 2005, `http://cm.bell-labs.com/who/akl/hash.pdf`.
9. J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner, *Property attestation -- scalable and privacy-friendly security assessment of peer computers*, IBM Research Report RZ3548, 2004.

10. Trusted Computing Group consortium, *Promotional literature and specifications available on public web site*, http://www.trustedcomputinggroup.org/.

11. Trusted Computing Group TPM Working Group, *TPM Main, Part 3, Commands, version 1.2 revision 62*, 2003, https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part3_Commands.pdf.

12. X. Wang, D. Feng, X. Lai, and H. Yu, *Collisions for hash functions MD4, MD5, HAVEL-128 and RIPEMD*, Cryptology ePrint Archive, Report 2004/065, 2004, http://eprint.iacr.org/2004/199.

13. X. Wang, Y. Yin, and H. Yu, *Collision search attacks on SHA1*, 2005, see http://202.194.5.130/admin/infosec/download.php?id=3.

14. _____ , *Finding collisions in the full SHA-1*, presented at the Crypto 2005 rump session, 2005, see http://202.194.5.130/admin/infosec/download.php?id=2.

15. X. Wang, H. Yu, and Y. Yin, *Efficient collision search attacks on SHA-0*, presented at the Crypto 2005 rump session, 2005, see http://202.194.5.130/admin/infosec/download.php?id=1.

Hagenbuchenweg 20, CH-8602 Wangen-bei-Dübendorf, Switzerland
E-mail address: jonathan.poritz@gmail.com