Cobweb: Practical Remote Attestation Using Contextual Graphs

Frank Wang*, Yuna Joung†, and James Mickens†
*MIT, †Harvard University

Abstract

In theory, remote attestation is a powerful primitive for building distributed systems atop untrusting peers. Unfortunately, the canonical attestation framework defined by the Trusted Computing Group is insufficient to express *rich contextual relationships* between client-side software components. Thus, attestors and verifiers must rely on ad-hoc mechanisms to handle real-world attestation challenges like attestors that load executables in nondeterministic orders, or verifiers that require attestors to track dynamic information flows between attestor-side components.

In this paper, we survey these practical attestation challenges. We then describe a new attestation framework, named Cobweb, which handles these challenges. The key insight is that *real-world attestation is a graph problem.* An attestation message is a graph in which each vertex is a software component, and has one or more labels, e.g., the hash value of the component, or the raw file data, or a signature over that data. Each edge in an attestation graph is a contextual relationship, like the passage of time, or a parent/child `fork()` relationship, or a sender/receiver IPC relationship. Cobweb's verifier-side policies are graph predicates which analyze contextual relationships. Experiments with real, complex software stacks demonstrate that Cobweb's abstractions are generic and can support a variety of real-world policies.

CCS Concepts • **Security and privacy → Trusted computing**; **Distributed systems security**; **Software security engineering**; *Usability in security and privacy*; • **Computer systems organization** → *Distributed architectures*;

Keywords Remote attestation, Trusted computing, TPMs

ACM Reference Format:
Frank Wang, Yuna Joung, and James Mickens. 2017. Cobweb: Practical Remote Attestation Using Contextual Graphs. In *Proceedings of SysTEX'17:2nd Workshop on System Software for Trusted Execution, Shanghai, China, October 28, 2017 (SysTEX'17),* 7 pages.
https://doi.org/10.1145/3152701.3152705

SysTEX'17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5097-6/17/10…$15.00
https://doi.org/10.1145/3152701.3152705

1 Motivation

Remote attestation allows a client machine to securely enumerate its executing software to a remote server. Once the server learns which code is running on the client, the server can decide whether the client should be trusted. For example, a server might only trust a client if the client runs a recent edition of the Linux kernel, and an up-to-date version of the Go runtime. Remote attestation is useful in distributed systems where participants reside in different administrative domains, or otherwise lack a priori trust relationships.

1.1 Remote Attestation in Theory

The textbook protocols for remote attestation [12, 19, 21] leverage trusted client-side hardware to bootstrap the attestation process. TPM chips [23] are the most common instantiation of such trusted hardware, and are deployed in commodity laptops, desktops, servers, and high-end embedded devices. A TPM chip is a tamper-resistant coprocessor that possesses a set of Platform Configuration Registers (PCRs); these registers are inaccessible to the normal CPU. Each TPM also possesses a public/private key pair. The private key is never revealed outside of the TPM. The public key is vouched for by a certificate that is signed by the TPM manufacturer (e.g., Intel).

When the client (i.e., the attestor) boots, the TPM resets its PCRs to a well-known initial state. The attestor's read-only firmware calculates a hash H of the first-stage bootloader, and invokes the TPM interface `extend(idx,val)`. This interface instructs the TPM to set `PCR[idx] = hash(PCR[idx] || val)`; the read-only firmware uses H as `val`, and uses an `idx` of 10 by convention [5]. Once the extend operation has completed, the read-only firmware jumps to the first instruction in the bootloader. The bootloader calculates the hash of the second-stage bootloader, and extends `PCR[10]` with this hash. The bootloader then jumps to the first instruction of the second-stage bootloader. This process continues as the machine loads the statically-linked kernel code, dynamically-linked kernel modules, configuration files, scripts, and user-level binaries. As the machine boots, the kernel maintains a list of `<file_name, hash(file_data)>` tuples representing the objects that have been used to extend `PCR[10]`.

Once the machine has fully booted, the value in `PCR[10]` is a succinct representation of the attestor-side software stack. The attestor contacts the remote server who will act as the verifier of the attestation. The verifier generates a nonce, and returns it to the attestor. The attestor then invokes the TPM's

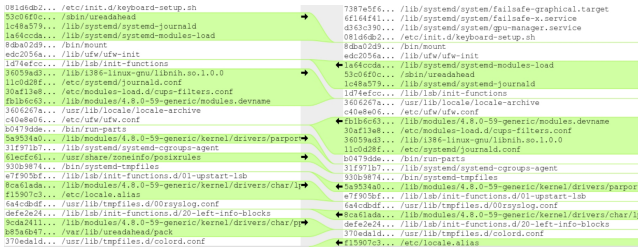

Figure 1. An example of the diffs between two IMA [5] logs that were generated by two boots of the same machine. The machine was a dual-core Dell ThinkPad laptop running Linux 4.8. The figure above shows differences in the load order of several kernel modules, shared libraries, and configuration files. **Each IMA log contained 2961 entries; the two logs had a Levenshtein edit distance of 1403.**

`quote(idx,nonce)` interface, passing 10 as the `idx` value. The TPM responds with a signed statement that includes the nonce and the `PCR[10]` value. The attestor sends this statement to the verifier, along with the list of `<file_name, hash(file_data)>` tuples. The verifier ensures that (1) the signature is from a trusted TPM, and (2) the reported value of `PCR[10]` actually corresponds to the cumulative hash of the reported objects. Finally, the verifier determines the trustworthiness of the attestor's software stack, using a database of trusted hashes to check whether (3) all (or an "important" subset) of the attestor's objects correspond to trusted ones.

1.2 Remote Attestation in Practice

Determining if condition (1) is true is straightforward—the number of TPM vendors is small, and a verifier can easily find and cache the public keys for the trusted vendors. Condition (2) is also trivial to check, since the verifier merely needs to perform several hash operations. However, determining whether condition (3) holds is difficult in practice. The trustworthiness of an attestor-side program is often *context-sensitive*, with different verifiers defining "context" in different ways. Thus, a simple database of trusted hashes is often radically insufficient to establish if an attestor should be trusted. The official TPM specifications are intentionally agnostic as to how verification policies should be defined and implemented [19], but defining and implementing policies are of great practical concern to system designers who wish to use remote attestation. Consider the following examples:

Nondeterministic load orders: As shown in Figure 1, attestors can load software components in a nondeterministic order. Even on a uniprocessor machine, load orders can be randomized by nondeterministic completion times for IO requests. Unfortunately, certain load orderings may trigger concurrency-related security bugs [26], as different processes race (for example) on kernel state involving memory management [25] or the file system [14]. For a verifier to detect these security problems, attestors need the ability to create richer attestation messages that go beyond mere hash lists, and include timing information at the granularity of system calls or other security-related operations. Verifiers need a way to define analyses over the resulting temporal graphs.

Information flow: A verifier may tolerate nondeterministic load orders, but reject certain information flows between attestor-side programs. For example, a verifier may want attestors to enforce SELinux-style mandatory access control, with low-integrity programs unable to feed input to high-integrity ones through the file system or IPC [4]. In this scenario, the verifier must ensure that the attestor runs a MAC-enforcing kernel, and *also* has defined a reasonable policy configuration. Thus, attestation reports must contain the contents of the policy files, not just their hashes, since the policy files may contain site-specific information that the verifier cannot anticipate.

A verifier may also want to examine *dynamic* information flows on the attestor. For example, the attestor may run a framework like CamFlow [13] that tracks data as it traverses OS abstractions like the file system and pipes. A verifier may want to audit provenance records to ensure that the verifier will not consume data from untrustworthy sources. The extant TPM specifications for remote attestation [19] are ill-suited to capture these kinds of constraints.

Dynamic state: Traditional attestation only examines the static content of attested objects. However, verifiers may also wish to validate dynamic program state. For example, in LKIM [8], a kernel's in-memory data structures are summarized at the time of attestation using functions which walk those data structures; depending on the functions, the summary data may be more complex than simple hash values.

Blacklists: A verifier may deem a particular object to be categorically unsafe, regardless of when attestors load the object. For example, a verifier might reject attestors that run a known-insecure version of a web browser. A verifier may also want to perform contextual blacklisting, e.g., to only allow an attestor to run sensitive corporate HR programs if the attestor runs trusted firewall software. Blacklisting is an important aspect of attestation—for attestors that run complex software stacks, a verifier will often have no opinion about many of the individual components, but will need to explicitly whitelist *or* blacklist a subset of the components.

Hash ambiguity: An attestor may not wish to precisely identify certain components in its software stack. For example, suppose that an attestor runs a Python interpreter. The attestor may only wish to reveal that it runs version 2.7.13, 3.4.6, or 3.6.2 of the CPython implementation. By maintaining ambiguity about the components in its software stack, the attestor makes it harder for a malicious verifier to directly exploit zero-day vulnerabilities in attestor code. The difficulty increases as the number of ambiguous components grows, since the attacker must reason about a larger number of potential cross-component interactions.

To support hash ambiguity, attestors must be able to launch "believable" versions of software components that

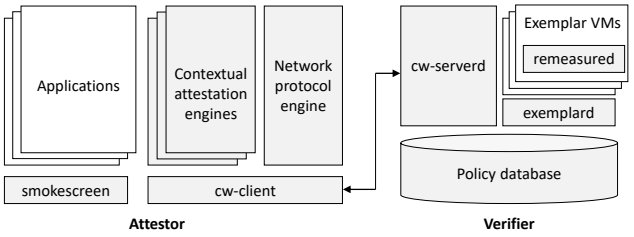

Figure 2. The Cobweb architecture.

the attestor does not actually use. Verifier-side policies also need a way to handle attestation messages which describe a variety of software components that the attestor *plausibly* runs. Reasoning about hash ambiguity is unsupported by the TPM specifications for attestation messages [22] and verifier-side hash databases [20].

Keeping policies fresh: A final practical concern involves verifier-side updating of polices. As the attestor's software is updated by vendors, the associated hash values and behavioral aspects change. Thus, a conscientious attestor that diligently updates its software will fail attestation if verifiers are not similarly diligent about updating policies. The TPM specifications are agnostic about how policies are updated, but automated tools for performing these updates are critical for making attestation usable.

2 The Design of Cobweb

The examples in Section 1 suggest that attestation in practice is a *contextual graph problem*. We now describe Cobweb, an attestation framework which leverages this insight to handle the complexities that arise in real-world attestation scenarios. Figure 2 provides a high-level overview of the Cobweb architecture.

2.1 Expressing Attestation Data

Cobweb defines a single attestation report as follows:

- At a minimum, an attestation contains a quoted `PCR[10]` value and a *classic attestation graph*. The classic graph is a list of `<file_name, hash(file_data)>` tuples which correspond to the objects that were cumulatively hashed to produce `PCR[10]`.
- In most scenarios, an attestation will also include a *contextual graph* which provides richer information about the attested software. In the contextual graph, each vertex represents a software component, and each directed edge represents a relationship between two components. Each vertex is associated with one or more *labels*; a label representing the hash of the associated object must be present, but attestors are free to attach arbitrary additional labels. Each edge is also associated with one or more labels which indicate the type of contextual relationship that the edge represents. For example, an edge might represent a parent/child `fork()` relationship, or an information flow via a pipe or a file.

Cobweb does not mandate that attestors collect predefined types of contextual data. However, for a given attestor/verifier pair, the attestor should collect at least the contextual information which is necessary to satisfy the verifier-side calculation of trustworthiness. In practice, this requirement means that verifiers ask clients to run a specific Cobweb attestation engine, as described below.

2.2 Attestor-side Infrastructure

Cobweb's `cw-client` program consists of two parts:

- The *contextual attestation engine* collects a specific set of information about the local software stack, and then bundles that information into a contextual graph. Engines are swappable, i.e., attestors can choose which engine is used by the local `cw-client`. Our Cobweb prototype predefines two different engines (although others are possible). Both engines use IMA [5] to implement `PCR[10]` extension. The first engine additionally tracks `fork()` and `exec()` relationships between processes [24]. In contrast, the second engine tracks dynamic information flows between different processes via OS interfaces like IPC [13].
- The *network protocol engine* serializes attestation reports using JSON, transmits those reports to verifiers using TLS, and responds to interactive queries that verifiers may issue during the evaluation of hit functions (§2.3).

`cw-client` is part of the attestor's trusted computing base, and should be checked by verifiers.

As mentioned in Section 1, an attestor may not wish to reveal the exact identity of certain software components. Instead, the attestor may want to state that it runs one of N possible versions. If the verifier trusts *all* of the possible versions, the verifier accepts the attestation as trustworthy. To enable this scenario, attestors use a Cobweb-provided launching service named `smokescreen`. For example, suppose that an attestor wants to serve web pages using Apache, while providing ambiguity about whether Apache, NGINX, or Tornado is actually used. To do so, the attestor installs all three web servers. The attestor then configures `smokescreen` to launch all three web servers, but to sandbox the latter two using mechanisms which are not exposed via attestation records; our `smokescreen` prototype for Linux uses namespaces [3, 6] to prevent the distractionary applications from actually interacting with the outside world. Thus, `PCR[10]` and the list of hashed software components will indicate that all three web servers are running, but a verifier cannot tell which are truly active.[1] Note that `smokescreen` is part of the attestor's trusted computing base, so verifiers must check whether attestors run up-to-date versions of the program.

[1] To disambiguate which web server is running, a curious verifier can try to probe attestor software at the application layer, e.g., by sending an HTTP request to the attestor and looking for a `Server` HTTP header. Preventing such application-level leaks is outside the scope of this paper.

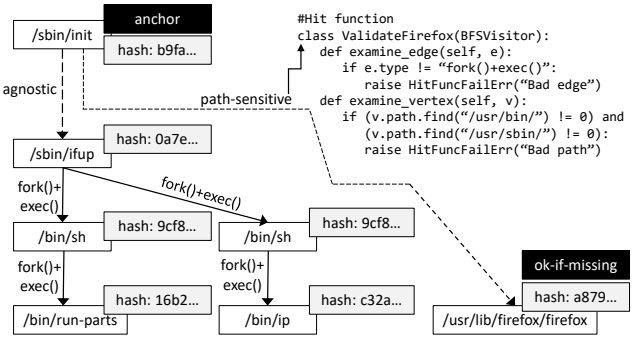

Figure 3. An example of a simple template graph. The template requires a matching attestation graph to have a particular version of `/sbin/init`, and a specific process tree rooted at `/sbin/ifup`. If a matching graph launches `/usr/lib/firefox/firefox`, then the launch must reside in a process tree whose ancestors are executables living in `/usr/bin` or `/usr/sbin`; this policy ensures that a web browser has not been launched by a suspicious executable that (for example) resides in a user's home directory.

2.3 Verifier-side Infrastructure

Defining policies: A verifier defines a policy using one or more *template graphs*. A template graph is a (typically proper) subset of an attestation graph that the verifier associates with a trustworthy software stack. At a high level, a template graph consists of several connected subcomponents; the template graph matches a provided attestation graph if there is a mapping from the template to the attestation. Figure 3 shows an example of a template graph. Below, we describe the mapping primitives that Cobweb provides.

Each concrete subcomponent of a template graph is stitched together using meta-edges. An *agnostic meta-edge* matches any path in the attestation graph to verify. A *path-sensitive meta-edge* is associated with a *hit function* that determines whether a path in the provided attestation graph is a match. Our Cobweb prototype allows hit functions to be defined in Python, using the graph representation defined by the `graph-tool` library [2]. A hit function returns a two-tuple. The first element is a boolean representing whether the hit function detected a match; if no match was detected, the second element contains diagnostic information about why the match failed.

A vertex in a template graph has one or more *labels*. At a minimum, a vertex is labeled with the hash of its associated software component. A template graph can have at most one vertex that possesses the special *anchor label*. An anchor label indicates that a matching vertex must come at the start of a matching attestation graph. Anchor labels are useful for defining policies which require a certain software component (e.g., `/sbin/init`) to start the attestor's boot process.

A vertex can also be associated with a hit function that determines if the vertex matches the one in the template graph. Hit functions can be used to perform context-specific analysis, e.g., to sanity-check the raw data for a vertex that represent a configuration file. To avoid clients having to unnecessarily send raw file data, the verifier fetches it on demand, validating the hash of the returned data before analyzing the raw data contents.

The `cw-serverd` program implements the verifier-side of the Cobweb protocol. A single policy is represented by one or more template graphs. A *required* template graph defines vertices and edges which must exist in the attestor-provided graph. A *forbidden* template graph specifies vertices and edges that must not exist in the attestor-provided graph. A template graph that lacks an anchor label can match in arbitrary positions in the attestation graph.

If `cw-serverd` determines that an attestation has failed, `cw-serverd` sends a summary of the failed graph predicates to the attestor. The attestor can then add, remove, or update the relevant software components.

Generating and maintaining policies: As vendors issue updates to attestor-side software, a verifier's policies become stale. Cobweb provides infrastructure to automate policy updating. The high-level approach is to run *exemplar VMs* which act as known-good versions of attestor-side software.

- `remeasured` is a daemon that runs inside the guest OS of a VM. `remeasured` uses platform-specific mechanisms (scripted using Python code in a configuration file) to fetch software updates for a VM. For example, on Linux, `remeasured` runs `apt-get update` followed by `apt-get dist-upgrade`. If the daemon finds updates, it installs them, reboots the VM, and then produces new attestation graphs for the machine.
- `exemplard` runs on the VM host, and manages a collection of exemplar VMs. In general, each VM is suspended to disk. `exemplard` periodically wakes up each VM, allowing the VM's `remeasured` to run. If `remeasured` generates new attestation graphs for a VM, then `remeasured` sends those graphs to `exemplard`. `exemplard` then runs the verifier's policy for that VM against the new attestation graph. If verification fails, `exemplard` notifies the administrator by sending an email, writing to an error log, or generating an HTTP POST request.

Importantly, `exemplard` maintains a longitudinal history of each VM's attestation graphs. These histories are useful if a verifier wants to define policies that accept "recent-enough" software stacks as valid. Also note that, if a verifier wishes to support attestor-side hash ambiguity (§2.2), then exemplar VMs must launch the relevant applications using `smokescreen`.

3 Implementation

Our prototype implementation of Cobweb runs on Linux-based attestors and verifiers. The implementation consists of 4,828 lines of Python code. The attestor-side library uses the IMA kernel module [5] to measure files and extend `PCR[10]`; the library also uses Linux's kernel-level event tracing [24]

to track system call behavior like invocations of `fork()` and `exec()`. The attestor-side library uses the CamFlow kernel module [13] to track cross-process data provenance via IPC and the file system.

On the verifier, Cobweb uses the `graph-tool` library [2] to execute the graph analyses that are associated with verification policies. `exemplard` uses QEMU [16] and `libvirt` [7] to manage VMs, and Berkeley DB [11] to store longitudinal attestation histories for exemplar VMs.

4 Evaluation

We use three case studies to evaluate Cobweb. In all scenarios, the attestor ran on a dual-core Dell XPS laptop with 2.40 GHz processors and 8 GB of RAM. The verifier was a dual-core Dell Precision desktop with 3.7 GHz processors and 16 GB of RAM. The attestor and the verifier were connected by a 1 Gbps LAN, to minimize network delays and focus on the computational overheads of attestation. In each scenario, the attestor contacted the verifier one minute after the attestor had rebooted. The descriptions below focus on the contextual graph used in each scenario; however, verifiers also ensured that the signed `PCR[10]` value agreed with the cumulative hash of the traditional IMA object list.

Standard Linux: In this scenario, the attestor ran a stock Linux 4.4.0 kernel. PCR extension was performed by the IMA kernel module. The contextual graph was a hash-annotated `fork()/exec()` process tree that was built using kernel-level tracing of `sched_process_fork` and `sched_process_exec` events [24]. The verifier's policy operated on the process tree, specifying an anchor vertex that represented a specific hash-version of `/sbin/init`. The policy required exact hash matches (but tolerated nondeterministic load orders) for 28 low-level executables and configuration files like `/bin/sh` and `/lib/systemd/systemd-*`. The policy also required exact hash matches and deterministic load orders for three process trees involving network state (`/sbin/ifup`, `/usr/sbin/NetworkManager`, and `/usr/lib/NetworkManager/nm-dispatcher`), and one process tree involving the launch of the Docker subsystem (`/usr/bin/dockerd`). The policy also required exact hash matches for the `ssh` and `nginx` binaries, and used hit functions to sanity-check `/etc/ssh/sshd_config` and `/etc/nginx/nginx.conf`. For example, the verifier ensured that `sshd` used at least 1024-bit server keys, and ran in privilege separation mode [15].

Static IFC policies: In this scenario, the attestor ran a Linux 4.4.0 kernel that used AppArmor [17], a mandatory access control system. AppArmor leverages per-application "profiles" to restrict application access to the network, the file system, and kernel capabilities like `CAP_SYS_PTRACE`. The contextual graph was a process tree, with the verifier requiring exact matches for 32 low-level system binaries and configuration files (without regard to their load order). The verifier also used hit functions to sanity-check the AppArmor profiles for MySQL, Firefox, Apache, libvirt, and Docker. For example, the `mod_apparmor` extension [1] allows individual Apache processes to transition between multiple AppArmor profiles, depending on the URL that a process is handling. The allowable transitions are statically specified by the profiles themselves; the verifier ensured that the transitions are reasonable, e.g., that no profile is allowed to serve data from sensitive directories, and that, for the directories which are being served, no profiles use blacklisted capabilities. Note that verifying these profile characteristics cannot be done simply by looking for profiles whose hash values are whitelisted. The reason is that the profiles on each attestor can vary according to the content that is served by the attestor's Apache installation. Thus, the verifier policy requires a hit function for each profile which performs semantic analysis of the AppArmor metadata in the profile.

Dynamic IFC polices: The attestor was a web server that ran a Linux 4.12.4 kernel with CamFlow [13] enabled. The attestor employed hash ambiguity (§2.2) to hide the precise identity of its web stack; using `smokescreen`, the attestor loaded an Apache/MySQL/PHP stack, an NGINX/PostgreSQL/Ruby stack, and a Tornado/MariaDB/Python stack. The contextual graph was a process tree that was augmented as we describe shortly. The verifier looked for exact matches for 51 system binaries (including those of the ambiguous stacks), and used hit functions to sanity-check various configuration files for each web stack. Another hit function examined `/etc/camflow.ini` to ensure that CamFlow tainted all untrusted user data in `/home/*`. The attestor augmented the process tree with additional edges which indicated 1) pairs of processes that had engaged in IPC, and 2) process/file pairs in which the file was tainted and the process read that file. The verifier checked the contextual graph to ensure the impossibility of tainted data flowing directly or indirectly to processes in a web stack. Note that the contextual graph did not disambiguate which web stack the attestor ran.

As shown in Table 1, Cobweb attestation generated a modest amount of network traffic. The majority of the traffic consisted of the gzipped attestation message that the attestor sent to the verifier; the embedded pathnames compressed well, but the hash values did not, since hash values are high-entropy. Verifiers had to fetch raw file data during the evaluation of hit functions, but the fetched files (e.g., `/etc/nginx/nginx.conf`) were small and text-based (and therefore highly compressible).

Table 1 also demonstrates that Cobweb attestation is fast, taking less than a second in all three scenarios. Figure 4 provides a more detailed view of the attestation costs.

- 53%–59% of the attestation time involved 1) the serialization and deserialization of Python data structures representing attestation graphs, and 2) the verifier-side

Scenario	Network traffic generated	Vertices	Edges	Hit functions	Hash ambiguity?	End-to-end attestation time
Standard Linux	129 KB	2059	2058	2	No	603 ms
AppArmor	137 KB	2149	2148	5	No	758 ms
CamFlow	150 KB	2590	3691	10	Yes	957 ms

Table 1. Summary statistics for the attestation case studies. The "Edges" and "Vertices" columns summarize the context-sensitive attestation graph, but ignore the traditional IMA-style list of hashed components; in all case studies, the IMA list contained roughly 1900 items.

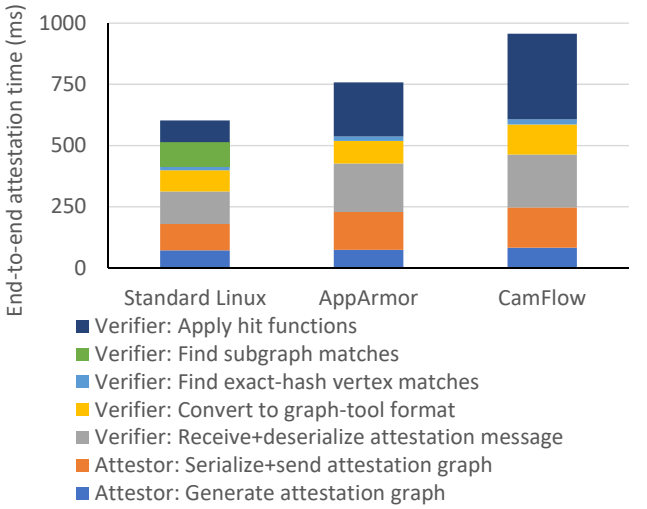

Figure 4. Cobweb's attestation costs. Not shown are the costs for invoking the TPM's `quote()` interface, and verifying the signature on the quote. These costs must be paid by any attestation system; on our test machines, the costs were 240 ms and 27 ms respectively.

conversion of those structures into a format that was compatible with the `graph-tool` library [2].

- Hit functions consumed 15%–37% of the total attestation time. The attestor and verifier were on the same LAN, so fetch overheads for file data were minimal; the bulk of the hit function cost involved the parsing and analysis of the file data.
- The first attestation scenario was the only one in which the verifier's policy looked for subgraph matches. Even though the policy only tried to match four subgraphs, the cost was relatively high (101 ms) because determining subgraph isomorphism is computationally expensive.

Given these results, we believe that Cobweb attestation is both expressive and practical.

As expected, the performance of `exemplard` and `remeasured` is dominated by the costs of VM spin-up and software update latency. For example, we installed `exemplard` and `remeasured` on a quad-core machine with 3.4 GHz processors, 32 GB of RAM, and a 120 GB SSD. Rebooting an Ubuntu VM with 4 GB of virtual RAM took roughly 7 seconds; checking for updates took roughly 3 seconds; if updates were found, installing those updates took a variable amount of time, but typically a few seconds. These aggregate costs were much larger than the time needed for `remeasured` to attest to `exemplard`.

5 Related Work

The Trusted Computing Group (TCG) has defined a high-level specification for remote attestation [19]. The TCG has also defined several low-level design documents for attestation reports [22], verifier-side hash databases [20], and a network-based attestation protocol [21]. Unfortunately, as explained in Section 1, these specifications lack the semantic richness that is needed for realistic attestation scenarios. The TCG specifications are also notoriously difficult to understand, despite the basic idea of remote attestation being simple. IBM's OpenPTS [9] framework implements the TCG attestation protocols, but to the best of our knowledge, the only open-source project which leverages those protocols is the StrongSwan VPN system [18] that forces clients to attest to a gateway before allowing network access. OpenPTS implements verification policies using simple finite state machines [10] which are too crude to capture policies like those in Section 4.

LKIM's measurement data templates [8] allow attestors to express richer information about OS state than mere hashes of kernel objects. However, LKIM is not a generic, end-to-end attestation framework; thus, LKIM has no solution for challenges like how to specify rich verification policies, and how to support hash ambiguity.

6 Conclusion

Cobweb is a framework that enables practical remote attestation protocols. These protocols are more complex than a textbook exchange of a list of hashes; these practical attestation protocols involve *contextual* information about the static and dynamic properties of attestor-side software. Using the generic abstraction of attestation graphs, Cobweb allows verifiers to define rich policies as graph predicates. Using hash ambiguity, Cobweb enables attestors to avoid precisely identifying their software stacks, while still allowing verifiers to determine that *some* trustworthy stack is running on the attestor. Cobweb also provides verifier-side infrastructure for automatically updating policies as exemplar software stacks receive patches or updates. Case studies involving three different attestation scenarios demonstrate that Cobweb is efficient, flexible, and easy to use.

References

[1] AppArmor. 2010. Mod apparmor. (December 10, 2010). http://wiki.apparmor.net/index.php/Mod_apparmor.

[2] T. de Paula Peixoto. 2017. graph-tool: Efficient network analysis. (2017). https://graph-tool.skewed.de/.

[3] Docker. 2017. Docker Overview. (2017). https://docs.docker.com/engine/docker-overview/.

[4] T. Jaeger, R. Sailer, and U. Shankar. 2006. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proceedings of SACMAT*.

[5] D. Kasatkin. 2017. Integrity Measurement Architecture (IMA). (2017). https://sourceforge.net/p/linux-ima/wiki/Home/.

[6] M. Kerrisk. 2013. Namespaces in Operation, Part 1: Namespaces Overview. (January 4 2013). https://lwn.net/Articles/531114/.

[7] Libvirt Project. 2017. The Libvirt Virtualization API. (2017). https://libvirt.org/.

[8] P. Loscocco, P. Wilson, J.A. Pendergrass, and C.D. McDonell. 2007. Linux Kernel Integrity Measurement Using Contextual Inspection. In *Proceedings of STC*.

[9] S. Munetoh. 2011. Open Platform Trust Services. (May 6, 2011). https://osdn.net/projects/openpts/wiki/FrontPage.

[10] S. Munetoh. 2011. OpenPTS Models. (December 11, 2011). https://github.com/openpts/openpts/tree/master/models.

[11] Oracle. 2016. Oracle Berkeley DB. (2016). http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[12] B. Parno, J.M. McCune, and A. Perrig. 2011. Bootstrapping Trust in Modern Computers. (2011). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/BootstrappingTrustBook.pdf.

[13] T. Pasquier, J. Singh, D. Eyers, and J. Bacon. 2015. CamFlow: Managed Data-Sharing for Cloud Services. *IEEE Transactions on Cloud Computing* (October 2015).

[14] M. Payer and T. Gross. 2012. Protecting Applications Against TOCTTOU Races by User-Space Caching of File Metadata. In *Proceedings of VEE*.

[15] N. Provos, M. Friedl, and P. Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of USENIX Security*.

[16] QEMU Project. 2017. QEMU: The Fast Processor Emulator. (2017). https://www.qemu.org/.

[17] E. Ratliff. 2017. AppArmor. (June 22, 2017). https://wiki.ubuntu.com/AppArmor.

[18] A. Steffen. 2011. The Linux Integrity Measurement Architecture and TPM-Based Network Endpoint Assessment. In *Proceedings of the Linux Security Summit*.

[19] Trusted Computing Group. 2006. TCG Infrastructure Working Group Architecture Part II: Integrity Management. (November 17, 2006). Specification Version 1.0, Revision 1.0.

[20] Trusted Computing Group. 2006. TCG Infrastructure Working Group Reference Manifest (RM) Schema Specification. (November 17, 2006). Specification Version 1.0, Revision 1.0.

[21] Trusted Computing Group. 2011. TCG Attestation PTS Protocol: Binding to TNC IF-M. (November 24, 2011). Specification Version 1.0, Revision 28.

[22] Trusted Computing Group. 2011. TCG Infrastructure Working Group Integrity Report Schema. (August 24, 2011). Specification Version 2.0, Revision 5.

[23] Trusted Computing Group. 2011. TPM Main Part 1: Design Principles. (March 1, 2011). Specification Version 1.2, Revision 116.

[24] T. Ts'o, L. Zefan, and T. Zanussi. 2017. Linux Kernel: Event Tracing Documentation. (2017). https://www.kernel.org/doc/Documentation/trace/events.txt.

[25] N. Wilfahrt. 2016. Dirty Cow: Vulnerability Details. (October 29, 2016). https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails.

[26] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. 2012. Concurrency Attacks. In *Proceedings of HotPar*.