Protecting Client Privacy with Trusted Computing at the Server

Current trusted-computing initiatives usually involve large organizations putting physically secure hardware on user machines, potentially violating user privacy. Yet, it's possible to exploit robust server-side secure hardware to enhance user privacy. Two case studies demonstrate using secure coprocessors at the server.

Alexander Iliev and Sean W. Smith
Dartmouth College

Trusted-computing (TC) initiatives potentially give large organizations ways to control individuals' use of their computers. Putting a physically protected component on a user's machine lets external organizations intrude on what previously had been the user's private space. However, we can turn the tables and put physically protected components on a large organization's machines—giving people some control over such organizations.

In this article, we describe our experiments in designing and prototyping TC at the server to enhance client privacy. Drawing on diverse tools ranging from oblivious RAM to switching networks to the Snort network intrusion detection system (NIDS), we have coded and tested these ideas on commodity secure hardware validated to the highest available level. Inspired by privacy problems in centralized directories in an enterprise public-key infrastructure (PKI), we designed and prototyped a privacy-enhanced X.509 certificate directory (www.ietf.org/rfc/rfc2459.txt) for a Dartmouth College-sized population. Inspired by privacy problems in encrypted archives of network traffic, we also built a system to ensure that access follows policy. (The code is available for download at www.cs.dartmouth.edu/~sasho/privdir/ and www.cs.dartmouth.edu/~pkilab/code/vault.tar.gz.)

Using TC at the server

Researchers have well documented the potential uses of TC to control users' behavior.[1,2] Popular uses are untamperable controls on electronic media usage (usually referred to as digital rights management [DRM]) and limiting which third-party programs can access data generated by a given program.

The flip side of large central entities having some jurisdiction over people's computers is that individuals can in turn control some aspect of the organizations' electronic systems in which they have a stake. We can achieve this control by using the same ideas that define TC, with trusted hardware at the organization's site. The implications for protecting users' privacy are considerable.

We call the physically protected and trusted component of a server K, after Franz Kafka's main character in *The Trial and The Castle*. (For the relevance of Kafka, and *The Trial* in particular, to the privacy debate, see Daniel Solove's article arguing that *The Trial* provides a better metaphor for the database problem than George Orwell's Big Brother.[3]) In any given client-server application, we can view K as an extension of the client: from a trust perspective, K acts on the client's behalf, but physically, K is colocated with the server. Both points are important. Clients can send any computation they would do themselves to K. Also, colocation with the server gives K a performance advantage and, even more importantly, the ability to be shared by many clients. This sharing makes economies of scale possible—K can be a more powerful and physically secure device than current TC initiatives seek to put on user machines.

To trust K so fully, clients must feel confident that K's computation is what they expected and that the server operator can't compromise or observe it—the server sees K as a black box, observing its inputs and outputs but none of the internal computation. (We elaborate on how a real instantiation of K attempts to be trustable in the next section.)

Another use of K is in securely storing sensitive user data. The Recording Industry Association of America (RIAA) and Motion Picture Association of America (MPAA) would like to use TC to control how individuals use data (such as music recordings). In applications where large organizations are storing user-sensitive data, individuals could use K to control how organizations use their data. One example is an archive of network traffic. The people whose traffic it contains might want to limit the ways in which organizations use it. Medical records of individuals' data stored electronically, for instance, need strong protection from arbitrary access by the people and organizations holding them.

Secure coprocessors

A secure coprocessor (SCOP) is generally defined as a computing environment, colocated at a host machine, which can safely house security-sensitive computation with some assurance. SCOPs even defend against physical attacks by the host operator.

A SCOP typically has general-purpose computer hardware in addition to some specialized additions, such as cryptographic acceleration hardware and tamper detection and reaction hardware. The classic example of a secure coprocessor is the IBM 4758 platform,[4] a PCI-attached device with an Intel 486 CPU running at 99 MHz with 4 Mbytes of RAM, 4 Mbytes of Flash memory, 16 Kbytes of battery-backed RAM, triple DES acceleration, and modular math acceleration for the RSA and DSA algorithms. It was the first device to be validated to FIPS 140-1 level 4. It's distinctly a server-oriented device, costing approximately US$3,000. However, other candidate devices are also available.

An important aspect of using SCOPs (as well as newer, smaller TC devices, such as the Trusted Computing Platform Alliance [TCPA], now the Trusted Computing Group [TCG] platforms) is that they can safely hold and use cryptographic keys that are somehow bound to a specific computational entity. In the IBM 4758, *outbound authentication*[7] binds private keys to a specific instance of software on a particular untampered device; in this case, the term *software* refers to all the programs executing in the SCOP—boot loader, operating system, and application. The SCOP security and certificate architecture ensures that use of these keys is confined to this software entity. If the SCOP is tampered with, or if the software is changed in unauthorized ways, the private keys are erased by the tamper-reaction mechanism or the firmware. Binding keys to program instances lets users trust that

- bits that they encrypted with a program *P*'s public key can only be decrypted by *P* running in an untampered SCOP, and
- bits signed by a program *P*'s private key must have actually been produced by *P* running in an untampered SCOP.

Potentially, newer TC devices can also be compelled to provide similar types of behavior.

PIR with secure coprocessors

One example of using a client extension inside a SCOP at the server site is realizing *private information retrieval* (PIR) of data from a server. PIR allows retrieval of information from a server, without the server learning anything about the data fetched. For example, we might want to retrieve a book description from Amazon.com, without Amazon being able to learn which book it was or to incorporate that request into statistics about book queries (for example, "everyone who asked about *X* also asked about *Y*").

We call this approach *hardware-assisted PIR*, or hw-PIR. (Earlier works referred to this approach as practical PIR[8,9] due to their motivation of being able to easily integrate PIR into the existing HTTP-over-SSL infrastructure of remote information access; we compare hw-PIR to the original PIR solutions in the "Previous work on PIR" sidebar.)

Our initial motivation for working on this hw-PIR server prototype was to add privacy protection to an X.509 certificate directory. PIR would remove the ability of the certificate directory to learn things about its users depending on which certificates they request. Other directories also have this problem—for example, the Attribute Authority in a Shibboleth-using organization is in a good position to observe and record where and when its members access online resources, which is unfortunate for a system that otherwise goes to great lengths to protect privacy. Dartmouth is interested in both these application areas because it has recently established a campus PKI and participated in Shibboleth pilots.

Consider the privacy implications if an adversary, Mallory, operates the certificate directory. Mallory learns of a secret message from Alice to Bob when Alice retrieves Bob's public key. She also learns of a signed message from Alice to Bob when Bob retrieves (or verifies) Alice's public key. Mallory thus has an easy way to do traffic analysis on secure communications, via the PKI at the endpoints. (See David Kahn's book *Codebreakers* for historical examples of traffic analysis.[10])

Further examples of where PIR can be useful abound, usually where traffic analysis of encrypted data can yield useful information. A medical doctor retrieving medical records (even if encrypted) from a database might reveal that the record's owner has a disease in which the doctor specializes. A company retrieving a patent from a patent database might reveal that they're pursuing a similar idea.

We note that anonymization solves an orthogonal problem. In the X.509 certificates scenario, anonymization would keep Mallory from knowing that *Alice* asked for Bob's certificate. Yet, Mallory would still know that

Previous work on privacy information retrieval

Theoretical computer scientists first formulated and solved the private information retrieval (PIR) problem without assuming the presence of trusted hardware. The initial solutions relied on the database being held on multiple noncommunicating servers,[1] which is a difficult assumption to assure in practice.

Eyal Kushilevitz and Rafail Ostrovsky first introduced the single server scheme,[2] which was followed by improved solutions in this setting. In current single server schemes, the server handles queries using only the cleartext database, so for every retrieval, it must read each item in the database (or else an adversary can learn that any untouched item is not the requested item). Thus, the server must perform linear work (currently, with considerable constant factors). On the other hand, the schemes have considerably sublinear (polylogarithmic) communication complexity between user and server—meaning that polylog(N) bits are exchanged to retrieve 1 bit from an N-bit database.

The scheme used in our prototype has a lineage of precursors:

- Oded Goldreich and Rafail Ostrovsky developed several algorithms for the oblivious RAM problem.[3] ORAM is concerned with hiding a program's pattern of memory accesses using a small trusted CPU and a large but untrusted RAM. This problem is to a large extent isomorphic to hw-PIR, which we address—memory locations correspond to records and the trusted CPU corresponds to the SCOP (to whom users whisper their requests over a secure channel).
- With a colleague, one of the authors[4] of this article pioneered using secure coprocessors for hw-PIR, with the aim that clients using existing network protocols like HTTP and SSL could also use them—that is, the client performs no other work besides negotiating a session key in the standard SSL manner, and no parties other than the server are involved.
- The hw-PIR work of Dmitri Asonov[5] aimed to improve the average response time offered by the Smith and Safford scheme, which was linear in the database size. (This was particularly problematic because the relatively slow SCOP had to do an amount of work linear in the database size, although this work could be divided across several devices.) Asonov suggested using the ORAM square-root algorithm for hw-PIR.

References

1. B. Chor et al., "Private Information Retrieval," *J. ACM*, vol. 45, 1998, pp. 965–982.
3. O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, 1996, pp. 431–473.
2. E. Kushilevitz and R. Ostrovsky, "Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval," *Proc. IEEE Symp. Foundations of Computer Science*, IEEE CS Press, 1997, pp. 364–373.
4. S.W. Smith and D. Safford, "Practical Server Privacy Using Secure Coprocessors," *IBM Systems J.*, vol. 40, no. 3, 2001, pp. 683–695.
5. D. Asonov and J.-C. Freytag, "Almost Optimal Private Information Retrieval," *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, eds., LNCS 2482, Springer, 2002, pp. 209–223.

someone was sending a private message to Bob, which might be interesting to her. If the domain of possible senders is small, Mallory can learn even more.

In contrast to preexisting schemes for solving the PIR problem, our scheme uses trusted hardware at the server and a preprocessed version of the database to service retrievals. These extra provisions reduce both communication and work, at the expense of servers having to install, and users having to trust an additional party (the SCOP).

Modeling retrieval

We can model the retrieval problem this way: the host contains a data set of N records each of size R, with unique names. (If records are not all the same length, they need to be padded. The pad contents are up to the application.) The records can be encrypted with K's public key so the host can't see their contents (who encrypts them and how they do it is beyond the scope of this article), or they can be in the clear. K has only enough memory to hold a small, fixed number of records at a time, so all other storage is on the host. Remote clients send queries for records by name, using an encrypted channel to K. The aim is for K to retrieve a requested record from the host, while hiding its identity from the host, which can observe all record I/O operations performed by K.

Modifications to the data set are outside this article's scope—they can be done before the data set is fed to the algorithm. For example, an administrator of an X.509 directory might update expired and reissued certificates before the PIR session begins.

Overview of the algorithms

We use the square-root algorithm introduced in the oblivious RAM (ORAM) work. For these algorithms, the records must be contiguously numbered from 1 to the database size N—prior hw-PIR/ORAM systems assumed that the client asked for a record by number. (We discuss how we make this more usable later on.)

The scheme proceeds in sessions, each serving up to I retrievals, for some I, where each session consists of

Randomly permuting the contents of records 1 through N. First, K encrypts each record in the database. Then, K uniformly at random selects a permutation π of $[1..N]$ and relocates the contents of each record r, $1 \le r \le N$, to record location $\pi(r)$, changing the

encryption along the way as needed. (We elaborate on the encryption used during the shuffle later in this section.) The relocations must be done so that the host can't learn which permuted record corresponds to which input record after having observed the pattern of record accesses during the permutation. In other words, the pattern observed by the host is independent of π (or else the host could learn something about π).

Retrieving* l *records, one at a time. (In this section, we say *retrieve* to mean the client-initiated action of obtaining one record and *fetch* to mean the SCOP action of reading a record off the host.) The first retrieval in a session is simple: K retrieves record r_1 requested by a client by fetching $\pi(r_1)$ from the shuffled database on the host. The host doesn't know anything about π, so it can't learn what r_1 was. For $i > 1$, retrieval i in a session is more complicated. To retrieve record r_i, K could just fetch $\pi(r_i)$ from the host, but that would reveal to the host one of two things: that r_i is the same as r_j from some previous retrieval j, or that r_i is different from all the previous retrievals. To hide the relationship among retrievals within a session, every retrieval i after the first one requires K to refetch every record r_j already fetched in this session, in addition to fetching r_i. If r_i has in fact already been fetched, then a randomly selected untouched record is fetched instead. Thus, the desired record r_i will be among the i records fetched, so the client can get its data. The host will be none the wiser about the identity of record r_i and its relation to the records fetched earlier in the session.

The running time of the ith retrieval is $O(i)$. Thus, the deployer can choose I to provide an upper bound on retrieval time; the session ends when I retrievals have been handled, and we move to a new session with a freshly permuted database. If I becomes higher than N, retrievals degenerate into linear running time because the SCOP must fetch every record when servicing one retrieval. Oded Goldreich and Rafail Ostrovsky called this the *square-root algorithm* because they chose I to be $\sqrt{N}$, thus minimizing the amortized work per query.[11]

Our prototype

Figure 1 shows our prototype's design. K actually consists of three coprocessors: one to continuously produce shuffled databases, one to handle retrievals, and one to deal with the Lightweight Directory Access Protocol (LDAP) over SSL. We have not yet implemented the LDAP SCOP, so LDAP handling is done by the OpenLDAP `slapd` server (see www.openldap.org) running on the host and the `slapd` *shell backend*, which interfaces with our retrieval coprocessor.

For our implementation, we used IBM 4758 Model 2 SCOPs running Linux and installed on Linux hosts.

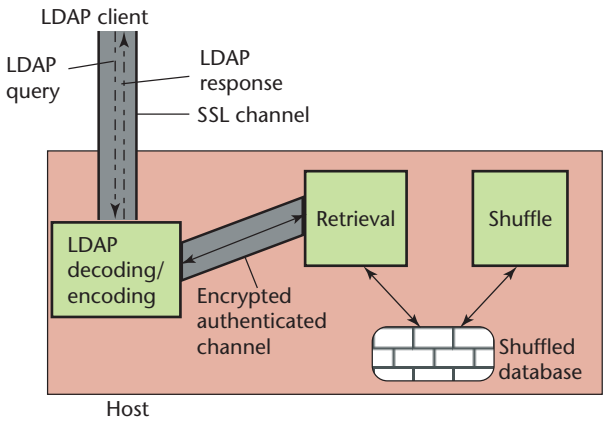

Figure 1. Hardware-assisted private information retrieval (hw-PIR) over LDAP system design. The system consists of three secure coprocessors: one to continuously produce shuffled databases, one to handle retrievals, and one to deal with the Lightweight Directory Access Protocol (LDAP) over SSL.

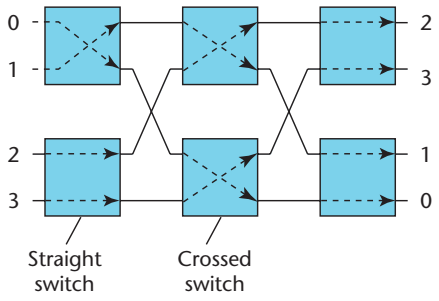

Figure 2. A Beneš permutation network with four inputs, performing the permutation ⟨2, 3, 1, 0⟩. The dashed lines represent switch settings, which depend on the permutation. The rest of the network only depends on the number of inputs.

Improving shuffling

We use Beneš permutation networks, a shuffling algorithm different from both previous suggestions of the square-root algorithm. A Beneš network can produce any permutation π of N items by performing $O(N \log N)$ switches.[12] A switch is the basic unit of a Beneš network, and it works on two items, either swapping their positions or leaving them as they were. The sequence of item pairs to be switched is independent of π—π only affects the switch settings. Figure 2 illustrates a small Beneš network.

To permute a database of records given a permutation π, K must follow these steps:

1. K internally computes all the switch settings to perform π. This can be done in $O(N \log N)$ time and

Other shuffling approaches

The oblivious RAM work uses Kenneth Batcher's sorting network, which is structurally similar to a Beneš network but consists of $O(N \lg^2 N)$ elements (comparators). Its advantage is that it can be performed using only $O(\log N)$ memory in the SCOP. Dmitri Asonov[1] suggested a $O(N^2)$ algorithm that consists of scanning the whole data set N times. It was the unusable running time of this algorithm—approximately three weeks on 10,000 records of 600 bytes each—that prompted us to find an alternative—the Beneš network.

Concurrently with our work, Asonov proposed another approach to shuffling the data set[2] that aims to reduce the number of I/O operations between the host and the SCOP. It creates *database slices*, each of which has one piece of a record. These slices are then shuffled under the same permutation and reassembled. This approach can reduce the number of communications between the host and SCOP from $O(N^2)$ to $O(N\sqrt{N})$, and because communications incur a high cost, it considerably improves the time needed by the original $O(N^2)$ algorithm.

References

1. D. Asonov and J.-C. Freytag, "Almost Optimal Private Information Retrieval," *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, eds., LNCS 2482, Springer, 2002, pp. 209–223.
2. D. Asonov and J.-C. Freytag, *Private Information Retrieval, Optimal for Users and Secure Coprocessors*, tech. report HUB-IB-159, Humboldt Univ., 2002.

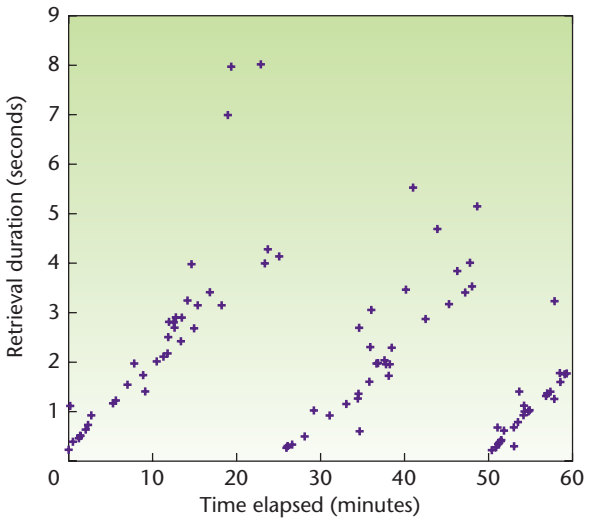

Figure 3. Retrieval times of an LDAP client. It was one of five independent clients, each sending randomly spaced queries, on average one every 40 seconds. The database size was 1,024 records, each about 1,500 bytes. (A Base64-encoded user certificate in the Dartmouth CA is 1,250 bytes.) The server was set up with one secure coprocessor (SCOP) continuously producing shuffled databases and onehandling queries. The increase in response time as a retrieval session progresses is clear, as well as the transition into fresh sessions.

space, although we're working on reducing the space bound. (Just storing an arbitrary permutation vector for n items requires $O(n \log n)$ bits space—n integers each of $\log n$ bits.)

2. K executes all the switches in an appropriate order—we do it in column-major order, going down each column of switches in order. K executes a switch by fetching the two records being switched, internally deciding whether to swap their locations, and then writing them out again. The host should not be able to tell whether the records were swapped or not; K can ensure this, for example, by changing the encryption key used. Thus, K uses a new encryption key for each column of switches (there being $2\lg N$ columns in total) but only needs to have two keys stored at a time for this purpose. (Straightforward techniques let K derive these keys from a master secret.) Also, K should use integrity checks during the procedure to immediately catch any attempts by the host to modify records or use old ones.

Thus, because π only affects the switch settings, and K ensures that the host can't deduce these settings, the host can't learn anything about π by observing the permutation proceedings. (See the "Other shuffling approaches" sidebar for related work in this area.)

Improving naming

Users typically want to ask for a certificate with a distinguished name, not with an abstract record number. In our design, K derives record numbers by hashing the record name. In our current prototype, we use hashing with chaining by using buckets of records; for this exposition, we ignored the hashing details. The main effect of the chaining is to introduce a runtime overhead of five to six times for the ranges of N we were considering.

We're looking into alternative approaches without this overhead, such as *perfect hashing*, which can provide a compact 1–1 function from a known set of N strings to the integers from 1 to N (see for example http://burtle burtle.net/bob/hash/perfect.html).

Results

We tested the prototype by running it continuously for some time and by having distributed LDAP clients (`ldapsearch` from the OpenLDAP package) send

queries periodically, asking for randomly picked legal names. Figure 3 gives the results, which indicate that the system can handle one query every 8 seconds, with a service time of 8 seconds in the worst case.

Because database shuffling is the bottleneck of this system, Figure 4 shows the running times of a single shuffle against two parameters: database size and record size.

Ongoing work

Currently, we are further mining the ORAM literature, extending our design and prototype to let users privately update records and reducing the internal SCOP memory requirement from its current $O(N \log N)$ bits (for calculating the Beneš switch settings).[13]

We have not experimented with the more complicated polylog solution to the ORAM problem given by Ostrovsky, because it has more overhead that the square-root algorithm for the ranges of N we could handle. It will be superior for larger N (the crossing point appears to be about a half to 1 million records) than we're currently considering and thus might be useful if higher-performance secure hardware becomes available.

Dmitri Asonov suggests the potential for an $O(N)$ shuffle by slicing records and making use of the fact that if all N pieces of a slice are inside the SCOP, they can be shuffled in linear time. This shuffle, however, depends on the record size; it might have large overhead because of having to encrypt and decrypt many small blocks (smaller than 64 bits), thus failing to take advantage of bulk cryptography hardware. Experiments seem necessary to establish how it performs.

Armored archive of network traffic

In private information retrieval, K protects the details of a client's request from the server. In contrast, in our armored data vault project,[14] we used K to ensure that the server's use of sensitive data abides by the client's policy, which is bound to the data when its created. The data we deal with here is archived network traffic, which can be sensitive to the people who generate the traffic and, thus, must be protected from inappropriate access.

Our prototype follows from the Packet Vault project at the University of Michigan's Center for Information Technology Integration (CITI),[15] which examined the engineering question of how a central authority, such as law enforcement or university administration, might archive traffic on a LAN for later forensic use. The packet vault stores all traffic flowing on a LAN and encrypts it to protect against unauthorized access. Subsequent discussions with Charles Antonelli and Peter Honeyman at CITI led to using a secure coprocessor to armor the Packet Vault, thus improving the assurance offered by the system.

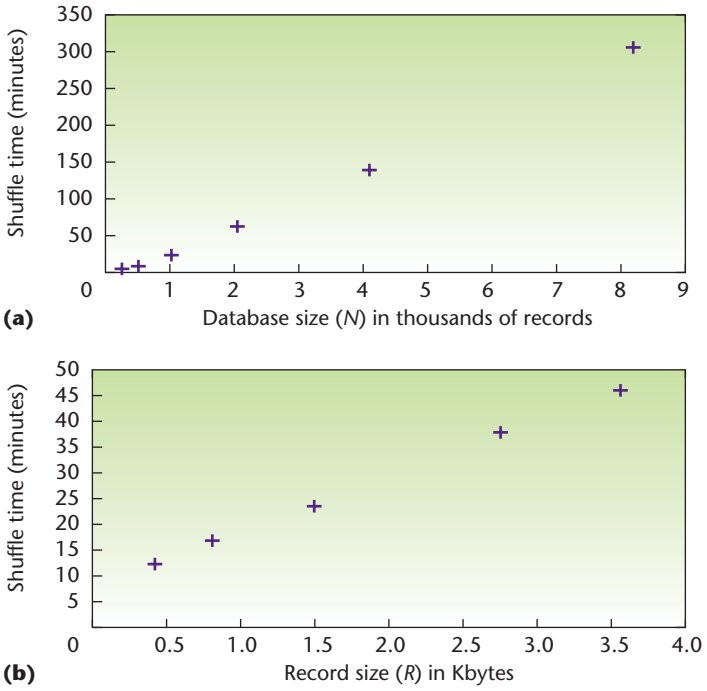

Figure 4. Running times of a single database shuffle, against (a) database size N, with record size fixed at 1,500 bytes and (b) record size R, with database size fixed at 1,024 records.

In particular, the Packet Vault's archives are encrypted by a master asymmetric key held by the highest authority in the system (such as university trustees), and this authority arbitrates all access requests, ideally following an explicit policy. The possibility thus exists for insider failure, perhaps indirectly through master key compromise.

Benefits of armoring

By armoring the archive—having the archive master key held inside a SCOP running a retrieval engine and policy checker—we can ensure that the archive can only be accessed via the SCOP and, thus, according to the policy. In the usual SCOP manner, the master key will be destroyed if the policy-checking software is maliciously modified or the SCOP is tampered with. This reduces the potential for circumventing the data protection system by social engineering or key compromise.

Using a secure coprocessor to arbitrate access to the traffic archive has advantages beyond providing high assurance that the policy must be followed. For example, the policy can specify

- that only some data function information is released, such as a statistical summary or sanitized version without IP addresses (the SCOP can safely perform this

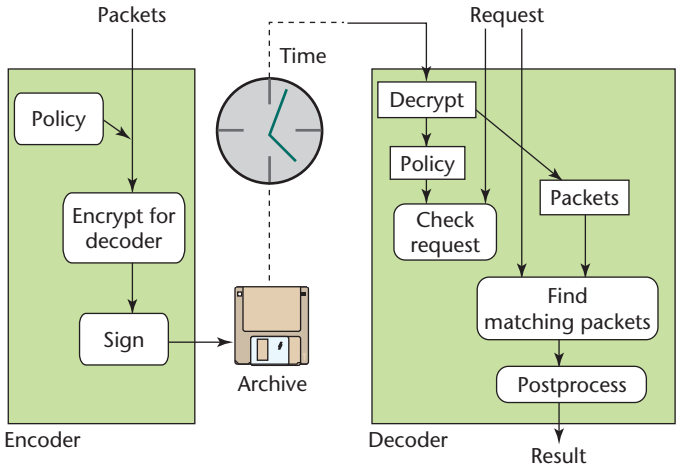

Figure 5. Armored vault system design.

function internally to prevent exposing the raw data);
- that an audit trail must be kept, which the SCOP can do, keeping an internal access log; and
- rate limiting, to make expansive searches through the archive (or fishing expeditions) more difficult.

In theory, adding a SCOP could provide these properties. But to show this could work in practice, we needed to actually demonstrate how a commodity SCOP can run a policy-controlled network traffic analysis tool.

System architecture

Figure 5 illustrates our system, which has two parts, an *encoder* and a *decoder*. The encoder makes the archives. It consists of a host and a SCOP that continuously collects the network traffic, encrypts it for the decoder, binds it to an access policy, and signs the whole lot to produce an archive piece sized according to the storage medium—a CD-ROM in the Packet Vault's case.

The decoder is a host and a SCOP that fields access attempts against the archive. Given an access request, it

1. checks the encoder's signature on the archive,
2. checks whether the access is allowed by the archive policy and if not, rejects the request,
3. finds the packets that match the request,
4. checks whether the matching packets violate the archive policy (for example, if too many packets matched),
5. performs any postprocessing specified by the policy such as IP scrubbing, and
6. signs and returns the resulting data.

Thus, a relying party can verify that the released data was obtained in accordance with the policy.

Prototype details

For our implementation, we used IBM 4758 Model 2 SCOPs running the production CP/Q++ OS and installed on Linux hosts.

Access policy. We gather all information relating to how archives can be accessed in an access policy. The policy is thus the central piece of the armored vault. The decoder will allow access to the archive only in accordance with the access policy, and no one can extract any other information because the coprocessor's design precludes circumventing its programmed behavior.

To represent access policies, we use a table with rows that represent different entry points into the data and columns that represent the parameters of each point. Anyone seeking access must select which entry point to use and then satisfy the requirements associated with it.

We define the parameters associated with an entry point in the policy as:

- *Request template*. This is a template of a query selecting the desired data, with parameters a particular access request must fulfill. We call the format of the query the *data selection language*. An example is, "All email to/from email address *X*."
- *Data subset*. This is a fixed expression in the data selection language that limits this entry point to some subset of all the archived data—for example, traffic between dates *A* and *B*. The query generated by the request template would then be matched only against data contained in this subset.
- *Macro limits*. This includes limits on various properties of the query's result, including the total number of packets or bytes in the result, how many hosts are involved in the resulting data, and the packet rate at the time of archival.
- *Authorization*. The authorization requirements for an entry point can include, for example, that the request must be signed by two district judges.
- *Postprocessing*. This procedure applies to the data chosen by the query to produce the final result to hand back to the requester, such as scrubbing all IP addresses or a statistical data summary.

To request data from the archive, a user indicates an entry point and provides parameters for its request template. The request contains the authorization data needed to satisfy the chosen entry point's requirements—for example, signatures from all the parties specified in the policy.

The actual policy must satisfy all legal queries while ensuring that no one, not even rogue insiders, can access more data or information than was intended. For example, failing to scrub IP addresses could result in an infor-

mation leak. The task of allowing a query only if it's legal is complicated by the fact that what is legal must be decided and expressed at the time of archival, while queries occur a long time after that.

Data selection language. We use an existing package, Snort version 1.7, to provide the packet selection capability in our demo. Snort is a `libpcap`-based NIDS. It selects packets using the Berkeley Packet Filter (BPF) language as well as its own rule system, which selects packets by their content as well as by header fields. This rule language is our data selection language. The Snort rule system is described in detail at www.snort.org/docs/writing_rules/chap2.html.

We chose Snort because it's an open-source tool in active development and use. Important features are IP defragmentation, the capability to select packets by content, and a developing TCP stream reassembly capability.

In a simple selection example, we can select to log TCP packets to the HTTP port with `log tcp any any -> any 80`. This only performs matching on packet headers. We can read it as "log TCP packets coming from any host, any port, going to any host, port 80."

A more complicated example (from the Snort Web site www.snort.org) uses content matching to produce an alert of a potential attack:

```
alert tcp any any -> 192.168.1.0/24 143
  (content: "|90C8 C0FF FFFF|/bin/sh";
  msg: "IMAP buffer overflow!";)
```

The content option is given, as are all options, inside parentheses.

Prototype results

Snort was one of the successful aspects of this prototype. Thanks to its capable packet detection engine and how well we could make it run inside the secure coprocessor, it will let us extend our policy capabilities considerably, especially when we consider application-level data selection and reassembled TCP streams become important.

High performance was not a goal of our prototype, but some sample figures illustrate the state of our project. The encoder could process a 1.6-Mbyte packet dump to produce an archive in 6 seconds. A 630-Kbyte dump took 2.3 seconds. The encoder performs everything described in our design, but being an early prototype, it's limited to processing only as much data as will fit into the coprocessor at once (about 1.7 Mbytes with our current prototype), so we have no numbers for larger packet dumps.

A decoder run on the 630-Kbyte archive (consisting of 1,000 packets), which selected 105 packets, took 6.3 seconds. Future optimizations will clearly have to focus on the encoder, which in practice will have to keep up with a fast network.

The community has long speculated about using secure coprocessors for computational enforcement of rights management. We carried this speculation one step further by completing an implementation on a commercial platform that could (in theory) be deployed on a wider scale without adding to the technological infrastructure.

A purported advantage of the coprocessor approach is the ability to insert a computational barrier between the end user and the raw data. To realize this, however, the barrier must be able to support useful filter computation.

Future work

An interesting area for future work would be to take some functionality out of the SCOP and put it onto the untrusted host. Classic work on database security has examined using a trusted filter in front of an untrusted relational database.[16] The aim was to keep the performance advantage of a relational database system and use a minimal filter to enforce access control.

For our armored archive prototype, minimizing the trusted component could involve extracting packet header fields so that an untrusted component on the host could handle retrieval by these fields, leaving the SCOP with the easier task of access control. Translucent database techniques[17] could protect header fields extracted in this manner.

On the other hand, the packet data could probably not be easily handled in this manner on the untrusted host because we have to support substring searches on the packet payload. Recent work on searching in encrypted data might be useful here.[18] The SCOP would then need to perform access control, query postprocessing, and translation of request data for the translucent database and encrypted-data search engine.

Clearly, practical secure hardware has some limitations, such as limited memory, that must be dealt with in a system's design. We have made progress in fitting as much as possible in these small embedded environments, but it's a safe bet that more ingenious designs than we have presented will further advance the usability and utility of SCOP-based privacy-enhancing systems.

Abstract ideas need to be concretely demonstrated if they are to have any impact on systems. We have presented two prototypes running in real secure coprocessors, thus hopefully bringing closer to reality actual use of such systems. □

Acknowledgments

This research is supported in part by the Mellon Foundation, the US National Science Foundation (CCR-0209144), Internet2/AT&T,

and the Office for Domestic Preparedness in the US Department of Homeland Security (2000-DT-CXK001). We also thank IBM Research for their assistance with the 4758 secure coprocessors. This article does not necessarily reflect the views of the sponsors. The authors are grateful to Charles Antonelli, Dmitri Asonov, Dan Boneh, Peter Honeyman, and the anonymous referees for their helpful comments. Preliminary versions of this material appeared at Privacy Enhancing Technologies 2002[14] *and the 2nd PKI Research Workshop, 2003.*[19]

References

1. E.W. Felten, "Understanding Trusted Computing," *IEEE Security & Privacy*, vol. 1, no. 3, 2003, pp. 60–66.
2. B. Arbaugh, "Improving the TCPA Specification," *Computer*, vol. 35, no. 8, 2002, pp. 77–79.
3. D. Solove, "Privacy and Power: Computer Databases and Metaphors for Information Privacy," *Stanford Law Rev.*, vol. 53, 2001, pp. 1393–1462; http://law.shu.edu/faculty/fulltime_faculty/soloveda/solove.html.
4. J.G. Dyer et al., "Building the IBM 4758 Secure Coprocessor," *Computer*, vol. 34, no. 10, 2001, pp. 57–66.
5. M. Bond and R. Anderson, "API-Level Attacks on Embedded Systems," *Computer*, vol. 34, no. 10, 2001, pp. 67–75.
6. S.W. Smith et al., "Validating a High-Performance, Programmable Secure Coprocessor," *Proc. 22nd Nat'l Information Systems Security Conf.*, Nat'l Inst. Standards and Tech., 1999.
7. S. Smith, "Outbound Authentication for Programmable Secure Coprocessors," *Proc. 7th European Symp. Research in Computer Science*, LNCS 2502, Springer, 2002, pp. 72–89.
8. D. Asonov and J.-C. Freytag, "Almost Optimal Private Information Retrieval," *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, eds., LNCS 2482, Springer, 2002, pp. 209–223.
9. S.W. Smith and D. Safford, "Practical Server Privacy Using Secure Coprocessors," *IBM Systems J.*, vol. 40, no. 3, 2001, pp. 683–695.
10. D. Kahn, *The Codebreakers: The Story of Secret Writing*, revised ed., Scribner, 1996.
11. O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, 1996, pp. 431–473.
12. A. Waksman, "A Permutation Network," *J. ACM*, vol. 15, no. 1, 1968, pp. 159–163.
13. A. Iliev and S. Smith, "Private Information Storage with Logarithmic-space Secure Hardware," *Information Security Management, Education, and Privacy*, Kluwer, 2004, pp. 201–216.
14. A. Iliev and S. Smith, "Prototyping an Armored Data Vault: Rights Management on Big Brother's Computer," *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, eds., LNCS 2482, Springer, 2002, pp. 144–159.
15. C.J. Antonelli, M. Undy, and P. Honeyman, "The Packet Vault: Secure Storage of Network Data," *Proc. Usenix Workshop Intrusion Detection and Network Monitoring*, Usenix Press, 1999, pp. 103–110.
16. R. Graubart, "The Integrity-Lock Approach to Secure Database Management," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 1984, pp. 62–74.
17. P. Wayner, *Translucent Databases*, Flyzone Press, 2002.
18. D. Boneh et al., "Public Key Encryption with Keyword Search," *Proc. Eurocrypt 2004*, LNCS 3027, Springer-Verlag, 2004, pp. 506–522.
19. A. Iliev and S. Smith, "Privacy-Enhanced Directory Services," *Proc. 2nd Ann. PKI Research Workshop*, Nat'l Inst. Standards and Technology, 2003, pp. 109–121.

Alexander Iliev *is a fourth-year graduate student of computer science at Dartmouth College. His research interests are in practical privacy for users of networked services, frequently making use of secure coprocessors and other secure hardware approaches. He has a BA in computer science from Dartmouth College. Contact him at sasho@cs.dartmouth.edu.*

Sean W. Smith *is an assistant professor of computer science at Dartmouth College and director of the Cyber Security and Trust Research Center at the Institute for Security Technology Studies. His current research and teaching focus on how to build trustworthy systems in the real world. He has a PhD in computer science from Carnegie Mellon University. He is the author of* Trusted Computing Platforms: Design and Applications *(Kluwer, 2005). Contact him at sws@cs.dartmouth.edu.*