Dartmouth College Computer Science Technical Report TR2001-400

An Armored Data Vault

Alex Iliev
sasho@cs.dartmouth.edu
Senior Honors Thesis. Adviser: Sean Smith.

June 1, 2001

Abstract

We consider the problem of secure long-term archiving of network traffic, an instance of the problem of storing private data securely. We approach the problem using secure hardware, which enables the enforcement of flexible access policy. The policy cannot be circumvented by anyone, even insiders, and so we are assured that access to the data is as originally intended. The policy can be expressed as any feasible computation, as it will be checked inside the secure hardware without possibility of interference. We discuss our design of a device to perform such network data archival and have implemented a prototype device. We discuss other possible application areas of the design.

1 Introduction

In this work we examine the problem of storing data with assurances as to how it will be used. We examine in detail the secure storage of network data, but one should keep in mind that many of the questions explored and techniques used are much more general. Flexibility in specifying allowed usage, and then enforcing the specified usage pattern are big questions, and network traffic storage provides a useful instance of these questions. Such storage has also become a hot topic lately, with much contention between parties wanting access to data, and those who do not want to have their data be seen.

The system we describe in this paper can take over an important part of the whole social process of controlling access to private data—enforcing the rules once they are in place. This job is arguably better left to a machine, the less likely to be disturbed the better. Having an impartial machine acting securely

as arbiter to stored data can leave policy-makers in the much improved position of being able to decide on policies which may be desirable but impractical for difficulty of enforcement.

The armored vault we describe is the beginning of a system which can potentially relieve the difficulties of trusting humans with one's private information. It is the next step from a security perspective after the Packet Vault, which began the exercise of cryptographically secure mass collection of net data.

Section 2 describes in some detail the motivations behind the Packet Vault and our Armored Vault. Section 3 describes our design, and Section 4 our implementation of the prototype. We conclude with a discussion of the relation of our work to the Advanced Packet Vault, the wider applicability of the design of our vault, and general future work.

2 Background

Here we examine some developments motivating our subject matter of comprehensive network traffic archives, and the utility of armoring.

2.1 Evidence collection

Examination and storage of network data has become an important discipline lately. In the US, controversy has been publicly raging since the revelation of Carnivore, a tool to wiretap network communication at ISP's. Carnivore is a software system designed to run at an ISP and collect communications of the parties under surveillance. Privacy advocates attack Carnivore on the grounds that such a system, once established and accepted, will allow the government to wantonly spy on individuals without their knowledge. They claim that even if the FBI acts in good faith, the system is technologically incapable of ensuring that data usable as evidence is collected in a minimal fashion—without obtaining information about parties not involved in the investigation. Some objections include: difficulties in obtaining the exact data which the recipient of the communication does, difficulties in keeping up selection algorithms with a highly loaded network, following assignment of IP addresses, and handling the general complexity of network protocols in order to extract useful information. All these accusations are of course speculations, as the FBI has not made the working mechanisms of Carnivore public.

An independent technical review of Carnivore [SHHPKM00] concluded that it is largely technologically sound, and will err on the side of under-collection

if the exact data needed is difficult to select from the network. They do give that this is dependent on correct configuration for the case in question though. There were high-profile followups to report though, like [BBF+00], which points out more questions and warns that there is a long way to go before Carnivore is ready for use.

The FBI defends Carnivore as being necessary to keep up with the Internet Age criminal. It claims that its use of Carnivore is exactly analogous to telephone wiretaps—only after authorization from a Federal judge, after which the collected data is subjected to the full scrutiny of the legal system.

To make the network archival question even hotter, law enforcement agencies in the European Union have pushed into consideration by governments their demands that all communication data be archived for extended periods—from [oPC00]:

> It is impossible for investigation services to know in advance which traffic data will prove useful in a criminal investigation. The only effective national legislative measure would therefore be to prohibit the erasure or anonymity of traffic data.

This kind of approach to chasing criminals on the Internet is potentially much more privacy-invading than the comparatively tame-looking Carnivore.

2.2 Complete archival of net traffic

Storing all network traffic is clearly not as wild an idea as one might think. Only from the law enforcement perspective it has many advantages over attempting to filter out the needed data on the fly and discarding everything else:

- As quoted above from [oPC00], authorities often do not know in advance where evidence may be picked up.
- Filtering for the needed data could be a complicated operation—reconstruction of several protocol layers, keeping track of dynamic IP assignments all take time. The speed of net traffic may be such that real-time filtering is unfeasible.

Moreover, traffic could be stored for other reasons, as research data for example.

2.3 Previous Work: The Packet Vault

The Packet Vault described in [AUH99] is the pioneering device to securely record all network traffic. It achieves the performance necessary to capture 10MB ethernet traffic, with ongoing work to accommodate 100MB networks too [AC00]. It attempts to store packets securely, so that they may be accessed only through the security mechanism imposed by the vault. The security mechanism is as follows: in an archive (on a CD-ROM) host-to-host *conversations* are encrypted with separate secret keys, and the set of these keys is encrypted using a public key algorithm (PKA) with the private key held by a trusted entity, the *overseers*. Access to the archived data is accomplished by getting the overseers to decrypt the secret keys for the desired conversations, which they presumably do once they decide all access conditions are satisfied. This approach leaves many questions unanswered, basically involving insider attack and flexibility.

2.3.1 Vulnerability to insider attack

The Vault depends on the trustworthiness of the overseers. If they are law-enforcement personnel for example, trusting a complete record of network traffic to them seems objectionable. It is not very different from letting them have a cleartext record of the net traffic, and trusting them to use it as all concerned parties (those who have data from/about them stored) would like. Even if the overseers act in good faith, their private key may be compromised, which could either expose all the archives on which the public key of the pair was used, or necessitate the archives' destruction.

2.3.2 Access flexibility

Accessing the archived packets is not very flexible. Full access or no access to some set of conversations can be granted to a requester; nothing else is possible. Some other useful access possibilities which the Packet Vault does not provide for are:

- Accessing data at finer granularity than a conversation. Rounding to the nearest conversation could either skip needed data, or over-select to the extent of making the released data unsuitable for evidentiary use. Simply using different secret keys for smaller chunks than conversations cannot solve this as we will eventually have to have a different key for every bit. A purely cryptographic solution cannot be complete, some computation has to take place to perform arbitrary selection of the data.

- Proof of the source of the released data. The only thing we have in the vault system is the assurance of the overseers that they authorized the access, at the alleged time. A simple extension (mentioned in [AUH99]) consisting of securely time-stamping and signing the data released could help here.

- Altering the packet data granted in some way, or giving output calculated using the archived packets and not the actual packets. Such a capability could be useful in a lot of applications—law enforcement may want to detect the presence of some party under investigation without needing the full record of their activities, system administrators could want to detect the presence of various attacks without needing to know the exact targets, researchers could want anonymous data for experiments (say with blanked IP numbers) or statistics of the net traffic.

  This particular feature cannot be provided by some simple cryptographic or other extension to the Packet Vault. By definition, computation must be done on packets which must not themselves be seen in their original form.

Notice that all the above features could be provided by trusting the overseers to perform them, for example perform calculations on the stored packets and give us results. Trusting refers not just to the integrity of the overseers, but also to the integrity of the machines on which they perform their computations. If these machines are compromised by intruders, the computation the overseers claim (and believe) to have performed could be maliciously modified.

3 Design

3.1 Overview

The previous discussion suggests a secure solution to the problem of providing more flexible access to stored archives—perform any computation needed inside secure hardware instead of trusting the overseers to do it. Moreover, we can examine the functions of the overseers in the current Packet Vault design:

- Hold an encryption keypair which effectively allows complete access to the archived data.

- (Potentially) hold a signing keypair to certify data they have let out of the archive.

- Make decisions about how entitled a requester is to receive the data they seek. This they do using whatever electronic *or* conventional means ("I know this guy") they have at their disposal.

These are functions which a trusted machine could perform just as well. The last point is the most difficult to translate to a machine, since we are moving into a purely electronic authorization domain, but this field is rapidly evolving and more and more solutions will be found.

With the above in mind, an initial design for combining the overseer functionality in the Packet Vault and the power which the ability to apply computation to our data can bring could be as follows.

- The overseers are simply replaced with a secure machine (call it Solomon) which possesses an encryption keypair and a signing keypair, a description of how to evaluate the "worthiness" of requests for data, and a description of what computation must be done to select and/or process traffic data to be given to a requester.
- The stored traffic is all encrypted with Solomon's encryption keypair. If we do not wish to trust someone to do this part, we could get Solomon to do it too.
- Requests for access to the stored data are given to Solomon, who can then (1) evaluate if the request is worthy of being honored, (2) compute what data is to be released and (3) perform whatever computation is needed on that data to produce the final result, which it then signs and releases.

Our design for an *Armored* packet vault, proposed in [SAH00], is a variation of the above theoretical beginning. It consists of two secure coprocessors–the secure machines–one for collecting the net traffic and producing archives, and the other for arbitrating access to the archives. We refer to them as the *Encoder* and *Decoder* respectively. We use two because their tasks could be separated by space and time, and also the job of the Encoder is high-load and continuous. The Encoder encrypts the stored traffic using the Decoder's encryption key, and signs the archive using its own signing key. The Decoder must decide if access requests against archives are authorized, decrypt the archive in order to perform selection of the desired data, compute the query result using the selected data (this computation could be just returning the selected data) and sign this final result. This is shown in Figure 1.

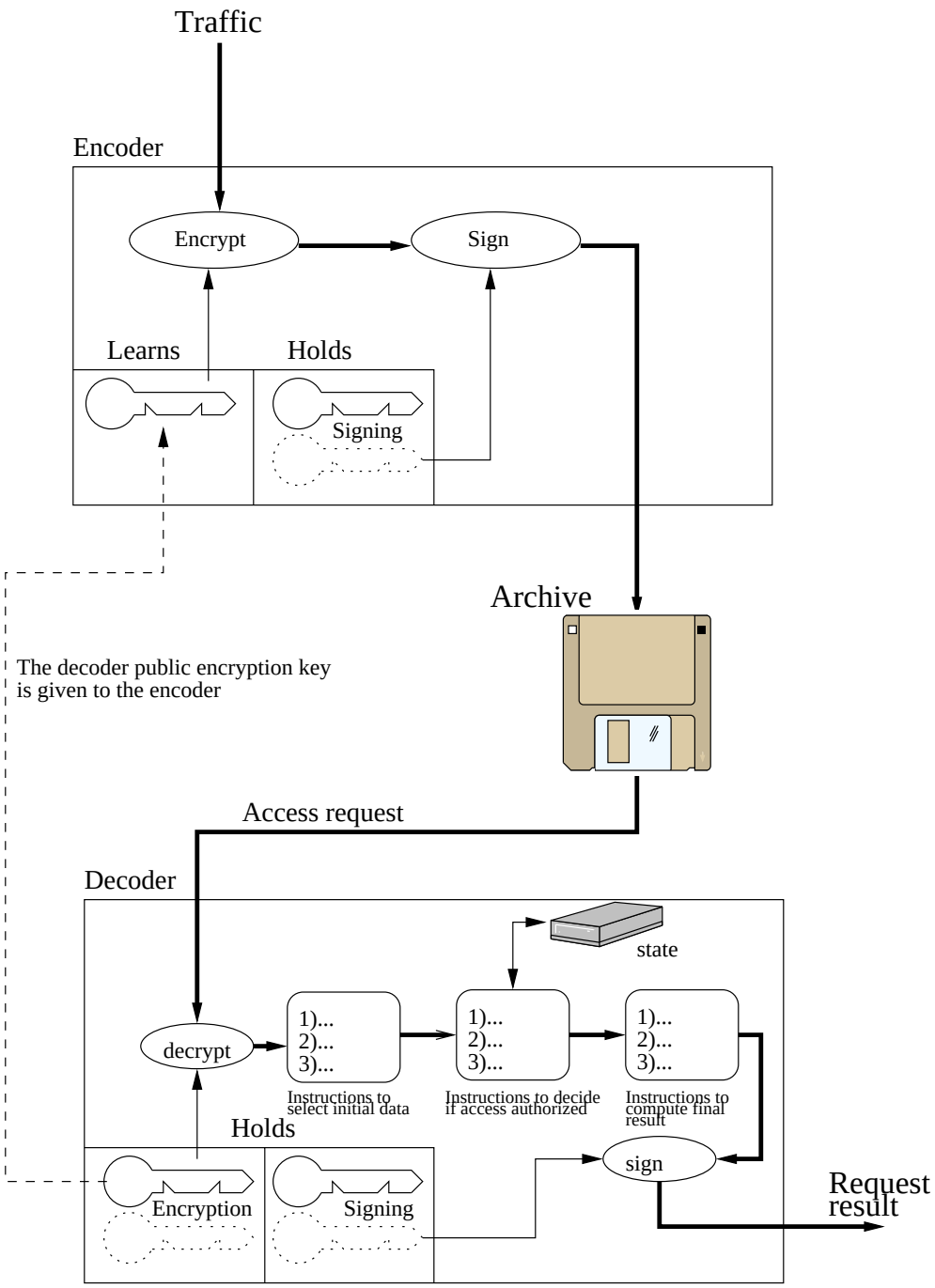

Figure 1: Overview of armored vault design. Details of the cryptographic organization are described in Section 3.4.

3.2 The secure hardware

3.2.1 Overview

The secure machine we use is the IBM 4758 programmable secure coprocessor [SW99, Smi01]. These coprocessors can

- Be programmed in C, with full access to an embedded OS (CP/Q++ from IBM) and hardware-accelerated cryptographic functionality.
- Carry out computation without possibility[1] of being observed or surreptitiously modified.
- Prove that some given data was produced by an uncompromised program running inside a coprocessor. This is done by signing the data in question with a private key which only the uncompromised program can know. Details of this process follow.

The coprocessors are realized as PCI cards attached to a *host* workstation. Currently drivers exist for Windows NT/2000, Linux, AIX and OS/2 hosts.

3.2.2 Outbound Authentication

The 4758 includes a mechanism for proving that output alleged to have come from a coprocessor application actually came from an uncompromised instance of that application. This is achieved by signing the output with an *Application keypair*, generated for the application by the coprocessor, and providing a certificate containing the public key of the signing pair, and a certificate chain attesting to the application certificate. In this chain is the identity of the application too. The structure of the chain is shown in Figure 2. To establish that the coprocessor application which produced some signed output is the expected one and is uncompromised:

1. Verify the signature using the public key in the application certificate.
2. Verify that all the certificates are signed as appropriate by their parent certificates.
3. Verify that the identity of the application, stored in the Layer 2 certificate, is what we expected. This identity includes things like program name, developer ID and program hash. The identity of the OS in Layer 2 can also be verified.

[1] as defined by FIPS 140-1 Level 4 validation

When this is done, we can be certain that the correct program produced the signed output. See [Smi01] for a comprehensive treatment of Outbound Authentication in the 4758.

3.3 Access Policy

We gather all the information relating to how archives are to be accessed into an *access policy*. The policy is thus the central piece of the armored vault. The Decoder will give access to the archive only in accordance with the access policy, and *no one* can extract any more information, since the design of the coprocessor precludes circumventing its programmed behavior.

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

3.3.1 Entry point parameters

The first two parameters define how the decoder will select the initial data of interest from the archive.

Request template This is a template of a query selecting the desired data, with parameters to be filled by a particular access request. We call the format of the query the *data selection language*. An example could be "All email to/from email address X".

Data subset A fixed expression in the data selection language which limits this entry point to some subset of all the archived data. This could be something like "email traffic only". The query generated by the request template would then be matched only against data contained in this subset.

The following two parameters define how the decoder decides whether a request is legal.

Macro limits Limits on various properties of the result of the query. These could be things like total number of packets or bytes in the result, how many hosts are involved in the resulting data, or packet rate with respect to time at the time of archival.

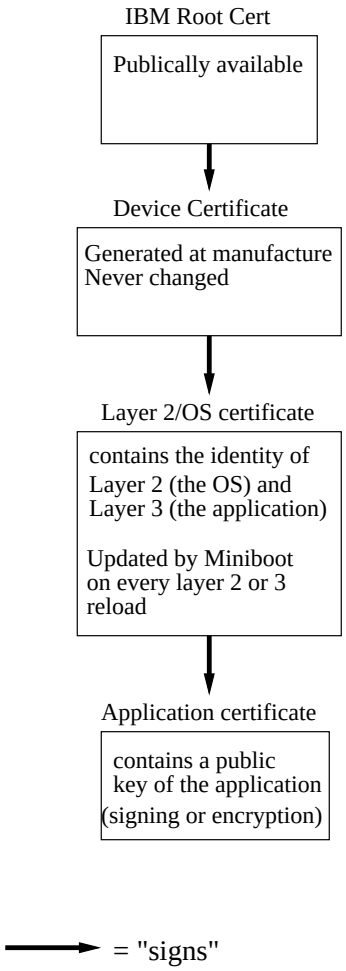

Figure 2: Simplified certificate chain attesting to an application keypair, and containing the application identity. *Layers* in the 4758 represent different stages in the startup process. Layer 0 is the first boot code in ROM (Miniboot 0), Layer 1 is the subsequent boot code (Miniboot 1) replaceable in Flash, Layer 2 is the OS, and Layer 3 is the application. Layers also represent a progressively tighter security environment, essentially with higher layers unable to observe or modify data in lower layers. When an application is loaded into layer 3, Miniboot 1, which handles the process, replaces the layer 2 certificate with a new one, which indicates the identities of the OS in layer 2 and the application in layer 3. Note that some certificates are missing from this chain, but their presence does not modify the usage of the chain described in Section 3.2.2

Authorization The authorization requirements for this entry point. These could be something like "Request signed by two district judges". See also Section 6.3.

This parameter defines how the Decoder will compute a final result for the query based on the initial data selected.

Post-processing A procedure to apply to the data picked out by the query to produce the final result to hand back to the requester. Things like "Scrub all IP addresses" or some kind of statistical analysis of the data.

The procedure for requesting data from the archive will be to indicate an entry point, and provide parameters for its request template. The request will contain the preliminary authorization data needed[2] to satisfy the requirements of the chosen entry point, for example be signed by all the parties who need to authorize the access.

The actual policy has to balance the opposing motivations of (1) enabling all legal queries[3], as decided at the time of archival, to be satisfied and (2) ensuring that there is no way for anyone, even rogue insiders, to gain access to more of the data than was intended. Note that "more" here means not just more bytes of data, but essentially more information about the archived data. For example failure to blank an IP address where it was intended to always be blanked represents giving access to more data than intended.

3.4 Cryptographic organization

The top-level cryptographic organization of the armored vault is as shown in Figure 1. Here we describe the details of each step.

The Encoder must be initialized by giving it the public encryption key of the Decoder which will control access to the archives produced by this Encoder. The key is contained in an Application Certificate, part of a chain as described in Section 3.2.2.

The Encoder produces an archive which is structured as shown in Figure 3. Encryption of the stored packets is two-level—first the packets are encrypted with a TDES session key, and then the session key is encrypted with the stored public encryption key of the Decoder. The Encoder provides verification for the archive as described in Section 3.2.2.

[2]The authorization environment in use could require followups to the outside by the Decoder.

[3]potentially far in the future

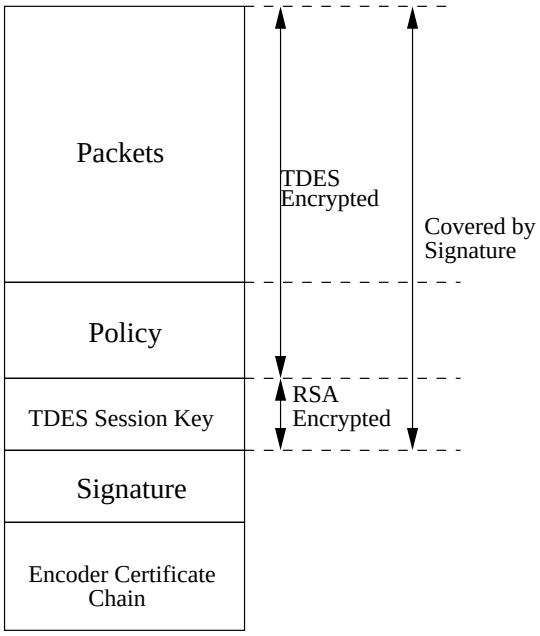

Figure 3: Structure of the archive. Note that the keypairs to encrypt the session key and to sign the archive are quite distinct.

When the Decoder receives a request against an archive, it verifies the archive by checking the signature and the identity and supporting chain of the Encoder which produced the archive. It then decrypts the session key using its private encryption key, decrypts the packet dump using the session key, and carries on with processing the request. When the final result is computed, it signs it in the same way that the Encoder signs an archive.

4 Implementation

4.1 Overview

Our setup consists of a Linux PC acting as the host to both the Encoder and Decoder cards. We picked Linux because we prefer open systems, and the Linux driver for the 4758 is open-source. See also Section 6.2. We used a PC with Windows 2000 to run the visual debugger for the 4758, connected to the cards with a serial cable. The visual debugger for Windows is more stable than the command-line debugger which exists for Linux, but otherwise equivalently powerful. An attempt at running the Windows debugger in Vmware on the Linux host was not entirely successful—the debugger was less stable than the one running on a real W2K installation.

The host-side user interface to the prototype vault consists of a command-line program which performs two tasks with the Decoder:

- Request its public encryption key/certificate together with a certificate chain (see Section 3.2.2) attesting to the key. For now the certificates are in the format used by the secure cryptographic coprocessor (SCC) interface of the 4758.
- Run a request for data against an archive previously made by the Encoder.

and two tasks with the Encoder:

- Set the encryption key and supporting certificate chain of the Decoder this Encoder is to work with.
- Produce an archive given a `libpcap`-format packet dump. This can only be done after a partner Decoder is established by supplying its public encryption key.

Packet dumps can be produced by any packet sniffer program which uses libpcap as its packet-capture mechanism (as libpcap includes a mode to just

dump the packets in binary form). Two possibilities are the quintessential `tcpdump`, and `snort`.

4.2 Encoder operation

The Encoder takes a `libpcap`-format packet dump and produces an archive, as shown in Figure 4. It performs everything described in our design, but being an early prototype is limited to processing only as much data as will fit into the coprocessor at once (1-2 MB).

4.3 The data selection language

We use an existing package, `Snort` [Roe99], to provide the packet selection capability in our demo. Snort is a `libpcap`-based NIDS which can select packets using the Berkeley Packet Filter (BPF) language as well as its own rule system which features selection by packet header fields (as does the BPF) as well as selection by packet content. This rule language is our data selection language. The snort rule system is described in detail at `http://www.snort.org/writing_snort_rules.htm`.

We chose Snort as it is an Open Source tool in active development, and active use. Important features are IP defragmentation, the capability to select packets by content, and a developing TCP stream reassembly capability.

4.3.1 Porting Snort

We had to compile a subset of Snort (essentially the packet detection system) to run inside the Decoder card and interpret requests for data. The full program includes many references to Unix syscalls, but the sections which did were irrelevant inside the card (for example alerts to a Unix-domain socket) and were macro-selected out.

We had to supply implementations for non-STDC functions which were still needed for packet detection and processing, like `inet_ntoa` and `getprotobynumber`.

The major challenge were the `stdio` functions to enable the transfer of data to and from Snort when it ran inside the Decoder. The coprocessor runtime does not provide the `stdio` functions involving `FILE` structs. This makes sense because the embedded OS, CP/Q++, has no filesystem as such. We implemented a "filesystem" by writing implementations for a few of the POSIX filesystem

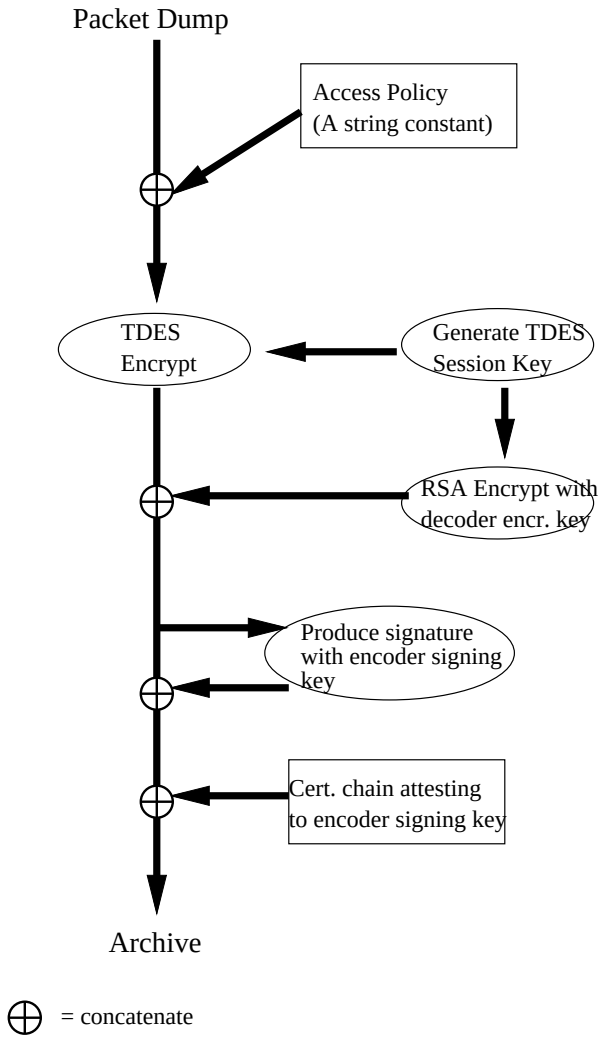

Figure 4: Encoder procedure, from a packet dump to an encrypted and signed archive.

calls (`open`, `write` etc.). These functions pass on the real work of the call to a module which is selected based upon the file's name. In C++ this would have been an abstract base class with methods like `write` and `close`, and concrete derived classes providing different implementations.

In this prototype we simply had files named `/membuf/*` be handled by a module which used a simple memory buffer to store data. During a Decoder run we put the packet dump in a buffer, set up a "file" called `/membuf/dump` ready to read from this buffer, and ran Snort telling it to look for the dump in `/membuf/dump`.

This filesystem mechanism can be extended to handle any type of IO we may wish to do, by providing new modules. We currently have a module to store files on the host, which will be a part of dealing with the very limited storage (RAM and Flash) in the coprocessor.

4.3.2 Snort rules

Snort rules specify what packets the NIDS selects for further processing (like logging or alerts). They select packets based on packet parameters, with options to match on any of the headers desired, as well as packet contents. A simple example to select TCP packets from the http port for logging is

```
log tcp any 80 -> any any
```

This only performs matching on packet headers. It could be read as "log TCP packets coming from any host, port 80, going to any host, any port". A fancier example (from the snort website) using content matching to produce an alert on noticing a potential attack is

```
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.

4.4 Decoder operation

The Decoder implements our design (limited to small archives which can fit in the coprocessor at once), with the following exceptions:

- No post-processing is currently possible. This will be a bit of work to implement fully, with reasonable capabilities, so we did not attack it in this prototype.
- The data-subset field of the access policy is ignored. This was for the rather mundane reason that we could have used Snort's BPF filtering capabilities here, but the BPF code was rather too Unix-entangled to be ported easily. Snort allows the usage of a BPF expression which is applied to all packets and only those passing the filter proceed to other processing, like selection via the Snort rules. This capability was functionally perfect for our data-subset requirements, but again some work to port.
- It has no authorization capabilites, except simple macro limits.

The detailed Decoder operation is shown in Figure 5.

4.5 Access Policy

We implement the access policy as XML-format text, with a *row* tag for the entry points (rows in the policy table), inside which are tags for all the entry point fields, with the exception of post-processing which we have not implemented yet. We do not have a document type definition (DTD) yet, and use very quick and simple "parsing" of the table. The table used in the current version of the prototype is shown in Figure 6.

4.6 Request Structure

Requests consist of a set of name=value assignments, one for the row number from the policy table through which the request is going, and the rest assignments for the request template of the chosen row. An example which could be used with our sample policy in Figure 6 is

```
row=1
port=80
```

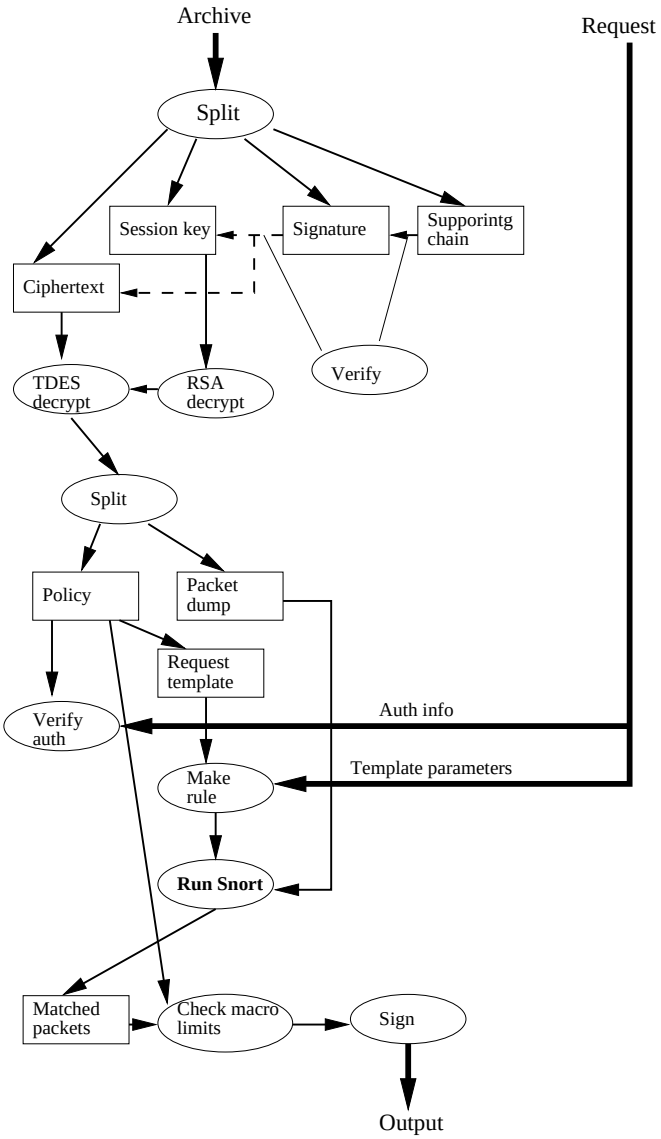

Figure 5: Decoder operation. The Decoder uses constant character strings inserted by the encoder between sections to perform all the splits. The separator strings are about 30 characters long, and the probability of them appearing randomly in the archive (and thus foiling this approach to sectioning) is negligible.

```
<policytable>

<title>
Experimental table
</title>

<row>

<reqtemplate>
log tcp any $port -> any any (content:"sasho"; logto:"snort.asc.out";)
pass tcp any any <> any any
</reqtemplate>


<datasubset>
</datasubset>

<macrolimits>
$total_packets < 100
</macrolimits>

<auth>
</auth>

</row>

</policytable>
```

Figure 6: Current prototype policy. This policy allows the selection of TCP packets containing the word "sasho", and coming from some port specified in the request (eg. "port=80"). If two or more packets match, the request will be declined. Note that the "logto" option in the Snort rule should really be added by the Decoder, as it is an implementation detail, not a part of the policy. The same applies to the "pass" rule, which has the effect of ignoring the remaining packets, unmatched by the "log" rule. Thus, once these Decoder manipulations are implemented, the "reqtemplate" element in this case would read `log tcp any $port -> any any (content:"sasho";)`

5 Discussion

5.1 Our Implementation

5.1.1 Snort

This was one of the successful aspects of this prototype. Snort is a very capable packet detection engine, and it runs happily inside the secure coprocessor. It will enable us to extend our policy capabilites considerably, especially when we start to consider application-level data selection, and reassembled TCP streams become important.

5.1.2 The policy

Our current prototype policy (in Figure 6) is fairly basic, but it does demonstrate many key points about the armored vault:

- Selection of packets can be computation-based. There is no way to select packets by content using differentiating cryptography[4] alone.
- Authorization decisions can be computation-based, and secure since they are running inside secure hardware. In this case, even in the absence of a PKI to perform full authorization, an authorization decision can be made based on the number of packets in the result—no query with more than one matching packet will be authorized. Since computation must be performed to calculate such properties of the matching data set, this cannot be done securely without using secure hardware.

We have not yet implemented post-processing of the initial matching packets, but this will be one of the next steps.

5.1.3 Performance

High performance was not one of the targets of our prototype, but we include some figures. The Encoder could process a 1.6 MB packet dump to produce an archive in 6 seconds. A 630K dump took 2.3 seconds. A 2.0 MB dump failed due to lack of memory (the 4758 model we use has 4 MB of RAM).

[4]meaning that different sections of the stored archive are encrypted with different keys

A Decoder run (without restrictions on packet number returned) on the 630K archive (1000 packets) which selected 105 packets took 6.3 sec. This figure is more indicative of real performance because a more complete version of the Decoder will use essentially the same approach to processing an archive as the current one. On the other hand, we will be working on the speed of the Encoder, especially for larger data quantities.

5.2 Relation to the Advanced Packet Vault [AC00]

One of the primary concerns of the Advanced Vault project is speed—to enable the vault to keep up with a 100Mb network at high load. The major areas of concern are system questions of keeping packets flowing into their final long-term storage as fast as they are picked up. Since we do not, nor do we intend to, consider questions of system infrastructure, our work can combine well with the Advanced Vault project to produce a fast and more secure device. The 4758 Model 2/23 secure coprocessor can perform TDES on bulk data at about 20 MB per second, which is sufficient to keep up with the Advanced Vault system infrastructure on a 100 Mb network.

5.3 Wider Applicability of the Design

The problem of storing network packets securely and with an attached access policy can be generalized to one of storing *any* data securely and with an access policy. Some concrete applications could be the following:

5.3.1 Remote data storage

We have close ties with the Condor Project at the University of Wisconsin in Madison[5], who have expressed an interest in remote storage of large data sets with associated access controls. This problem falls squarely into our system of securing data with flexible and assured access policy. If a researcher at site A wants to send data to site B, but ensure that the data is accessed only in accordance with some policy, she could proceed as follows. She uses a version of our Encoder to secure the data and attach her policy. She sends the data to site B, who have the Decoder to work with the data. They can gain only the kind of access site A specified. For example access could be limited to some specific research group, or the data can only be accessed a limited number of times, or some details about the data must always be hidden even if they are used in calculations inside the vault. One coprocessor may suffice in this

[5] `http://www.cs.wisc.edu/condor/`

scenario, and it would have a different user interface than the armored packet vault (operating via an SSL connection to a client program perhaps), but the basic idea of computationally-expressed access control executed inside secure hardware remains.

5.3.2 Academic PKI

An academic PKI is an area of interest for the Dartmouth PKIlab, and one of the questions that arises is what to do about confidential student data which may have to be disclosed under special circumstances. As an example, email on the school mail servers is considered confidential, but computing personnel have to provide access to it in the case of a legitimate law enforcement request. If students hold encrypted data in a PKI environment, there may be reasons to have access to it under special circumstances. Key Escrow is one solution, whereby the authorities have a master key, or know a trapdoor which allows access to the data in question. The possibility exists that the authorities may have a lapse of consciousness, or their equipment may let them down, resulting in undesirable (from the combined perspective of all parties involved) exposure of data.

If access to such student data is controlled by a data vault with an access policy, the exact circumstances under which data access may be granted can be spelled out, and will be adhered to.

6 Future Work

6.1 Immediate Work

We plan to expand our prototype along these major directions:

6.1.1 Performance

The limitation of requests and archives being able to fit inside the coprocessor is unacceptable, and will be lifted. We will augment the current arrangement to enable arbitrary-sized packet dumps and archives to be passed into the vault co-processors. Then archive size will be decided by the long-term storage medium, like CD-ROM.

Related is the ability of the Encoder to capture packets in real-time. First, we plan to attach the Encoder card to a packet source (some sniffer) to enable live data collection. See the next section for further work.

6.1.2 More Capable Policy

The capabilities of the policy mechanism are very limited currently. The query language (Snort rules) is very flexible, but the macro limits and authorization procedures are very bland. We plan to implement limits on more parameters like packet quantities/rates and number of hosts involved. On the authorization side, the first thing would be to implement history checking services of some kind so that decisions on allowing access now can be made based on accesses in the past. A full authentication of requests would depend on some identification/authorization infrastructure (like SPKI) and so we expand more on it in Section 6.3. For post-processing, we will implement scrubbing functions to erase any packet parameters required.

6.2 After the Immediate Work

As pointed out in [BBF+00], it may be necessary to consider selecting data at a sub-packet level, which would be a part of enabling access requests at the application layer rather than the network/transport layers as is the case now. `Snort` has a developing TCP stream reassembly functionality which will be very useful in this regard.

We plan to address the implementation difficulty of performing external-external[6] TDES encryption and SHA1 hash calculation in one operation on the 4758. External-external operation is needed to obtain the high speeds necessary for handling a high network load, but two separate operations–for encryption and hash calculation–would leave the data open to modification on the host before its hash is produced. Enabling this serialization of DES and SHA could involve modifications to the existing host device driver.

Another approach to the secrecy-authenticity procedure on the Encoder is the encryption mode described in [Jut00], which adds message integrity checking at the cost of $\log n$ extra block encryptions. This approach would have the drawback that the DES hardware on the 4758 may not be able to directly handle this new mode.

As described in the proposal [SAH00], the binding of an archive to a single

[6] Whereby the coprocessor DES hardware operates directly on buffers on the host (vie PCI), without the data ever being placed in coprocessor memory.

Decoder leaves the door open to denial of service attacks through Decoder destruction (just a tamper attempt will do the job). This binding is the current way of ensuring that the archive is accessed only through its attached policy, and there are no easy alternatives for achieving this assurance. Investigating the question will be important in making the armored vault practically acceptable.

6.3 Long term work

A serious part of a fully viable data vault system will be integration into an identification/authorization infrastructure, most likely some type of PKI. This would enable us to implement policy requirements like "authorization from three school officials ranked above the post of dean" by using what mechanisms the PKI offers for establishing posts/officers. Ongoing PKI work at Dartmouth could be very useful here.

7 Acknowledgments

This project was supported in part by Award No. 2000-DT-CX-K001 awarded by the National Institute of Justice, Office of Justice Programs.

The opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Department of Justice.

This work was also supported by Internet2/AT&T.

The author was supported in large part by Iguana.

I am grateful too to my adviser Sean Smith for guiding me through my first and not last journey into questions of computer security.

References

[AC00] Charles Antonelli and Kevin Coffman. Advanced packet vault. `http://www.citi.umich.edu/projects/vault-adv.html`, 2000. Project description.

[AUH99] C.J. Antonelli, M. Undy, and P. Honeyman. The packet vault: Secure storage of network data. In *Proc. USENIX Workshop on*

Intrusion Detection and Network Monitoring, Santa Clara, April 1999.

[BBF+00] Steven Bellovin, Matt Blaze, David Farber, Peter Neumann, and Eugene Spafford. Comments on the carnivore system technical review. `http://www.crypto.com/papers/carnivore_report_comments.html`, December 2000.

[Jut00] Charanjit S. Jutla. Encryption modes with almost free message integrity. Cryptology ePrint Archive, Report 2000/039, 2000. `http://eprint.iacr.org/`.

[oPC00] EU Working Party on Police Cooperation. Relations between the first and third pillars on advanced technologies, Nov 2000. ENFOPOL 71, REV 1.

[Roe99] Martin Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference - LISA '99*. USENIX, November 1999. `http://www.usenix.org/publications/library/proceedings/lisa99/roesch.ht%ml`.

[SAH00] S.W. Smith, C.J. Antonelli, and Peter Honeyman. Proposal: the armored packet vault. Draft, Sep 2000.

[SHHPKM00] Stephen P. Smith, Jr. Henry H. Perritt, Harold Krent, and Stephen Mencik. Independent technical review of the carnivore system. `http://www.usdoj.gov/jmd/publications/carniv_final.pdf`, Dec 2000.

[Smi01] Sean W. Smith. Outbound authentication for programmable secure coprocessors. Submitted, Mar 2001.

[SW99] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, 1999.