GnatDb: A Small-Footprint, Secure Database System

Radek Vingralek*

STAR Lab, InterTrust Corp.
4750 Patrick Henry Drive
Santa Clara, CA 95054
USA
vingralekr@acm.org

Abstract

This paper describes *GnatDb*, which is an embedded database system that provides protection against both accidental and malicious corruption of data. GnatDb is designed to run on a wide range of appliances, some of which have very limited resources. Therefore, its design is heavily driven by the need to reduce resource consumption. GnatDb employs atomic and durable updates to protect the data against accidental corruption. It prevents malicious corruption of the data using standard cryptographic techniques that leverage the underlying log-structured storage model. We show that the total memory consumption of GnatDb, which includes the code footprint, the stack and the heap, does not exceed 11 KB, while its performance on a typical appliance platform remains at an acceptable level.

1 Introduction

Many consumer appliances, such as mobile phones, cameras, portable music players, set-top boxes, personal digital assistants (PDA's) and smartcards, are equipped with an embedded microprocessor to allow fast upgrades and to reduce the appliance size [24]. The appliances frequently store and maintain valuable data such as passwords, private keys, health records or money. Therefore, it is important to protect the integrity of the data against accidental corruption resulting from a system crash caused by, for example, a power loss or a software bug. Most database systems protect data against accidental corruption by implementing transactional updates and performing frequent backups. Although most appliances support backups by synchronizing their local storage with a PC or a remote server, none of the appliances known to the author support transactions.

Many appliances need to protect data integrity not only against accidental corruption, but also against malicious corruption (*tamper-detection*) and unauthorized reading (*secrecy*). For example, a phone-smartcard user may obtain free phone calls by tampering with the data stored on the smartcard. Similarly, set-top box user may view TV channels for free by reading secret keys from the set-top box's storage. The research prototypes that provide secrecy and tamper-detection typically use a combination of symmetric key encryption and one-way hash trees (*Merkle trees*) [12, 9, 20]. (Nodes of a Merkle tree contain one-way hashes [14]. The internal nodes validate their children and leaf nodes validate data records. Data records are updated and validated by traversing a path in the tree.)

We use Digital Rights Management (DRM) systems as a motivating example of a system that requires secure and reliable storage of data with monetary value. DRM systems enable secure binding of digital content (such as software, music, video, e-books or email) to a *contract*. The contract is a program, which is executed each time the content is released to the user. Examples of contracts include "release the content after an up-front payment", "release the content for free up to n times" or "charge a fee for every release of the content or release the content for free if the user provides personal data". Execution of contracts frequently requires read and write access to persistent data, such as account balances, usage counters or digital certificates. The data needs to be protected against both accidental and malicious corruption. DRM systems typically need to store relatively modest volumes of

*Author's current address: Oracle Corp., 400 Oracle Parkway, Redwood Shores, CA 94065, radek.vingralek@oracle.com.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

data ranging from tens of kilobytes up to a megabyte.

In this paper we describe *GnatDb*, which is an embedded database system that provides both secrecy and tamper-detection. GnatDb is designed to run on a wide range of appliances, some of which have very limited resources. Therefore, its design is heavily driven by the need to reduce resource consumption. RAM is frequently the most critical resource either because the of a limited die space on single-chip devices, such as smartcards, or because large volumes of RAM are used for other purposes (such as stream buffering). Many appliances are built only with tens to hundreds of kilobytes of RAM available [23, 7, 22, 18]. Consequently, the design of GnatDb aims to reduce the code footprint as well as stack and heap memory consumption.

Instead of supporting full transactional semantics of updates, GnatDb supports only atomic and durable updates. It does not implement concurrency control, because most appliances either do not support multitasking or do not require concurrent access to the database. GnatDb implements update atomicity by leveraging the log-structured storage model [16], which we show is amenable to a simple, light-weight implementation and well-suited for the flash-memory-based storage found in many appliances. GnatDb integrates log-structured storage with its security primitives in a novel way that avoids construction of the Merkle tree and yet allows data validation in a constant time. We show that the total memory consumption of GnatDb, which includes the code footprint, the stack and the heap, does not exceed 11 KB, while its performance on a typical appliance platform running a DRM benchmark remains at an acceptable level.

2 Related Work

The design of GnatDb is strongly influenced by its predecessor, TDB [12]. Similar to GnatDb, TDB provides secrecy and tamper-detection. TDB also leverages log-structured storage organization and integrates it with data integrity protection. The major difference in the design of GnatDb and TDB stems from their target environments: while TDB is appropriate for PCs or similar devices (such as high end set-top boxes or game consoles), GnatDb is designed for appliances including single-chip systems with very limited memory. The lowest layer of TDB has a code footprint of 142 KB and the additional three layers add another 108 KB. Although the total code footprint is comparable to other embedded database systems, it by far exceeds the memory available in many appliances. Therefore, one of the major goals in the design of GnatDb is to provide the same level of protection as TDB and yet reduce the code footprint to less than 10 KB. Unlike TDB, GnatDb does not implement an object-oriented interface or collections. Its storage model does not scale to large numbers of records and its design is not optimized for performance. TDB implements the protection against malicious data corruption by integrating a Merkle tree with the location map that is used in log-structured storage systems. GnatDb, on the other hand, uses much simpler (and not scalable) location map organization and completely avoids building the Merkle tree.

A number of file systems protect secrecy of files by encrypting them with a secret key [1, 25, 4, 10]. Provos designed a similar protection for virtual memory pages [15]. Unlike GnatDb, such file systems provide only secrecy, but not tamper-detection. Several file systems provide also tamper-detection: Fu, Kaashoek and Mazieres designed Read-only File System, SFSRO, which embeds a Merkle tree in the inode hierarchy [9]. The root hash, which certifies the integrity of the file system, is signed by the file system's owner. Stein, Howard and Seltzer designed Protected File System, PFS, which is layered on top of a write-ahead file system [20]. PFS validates blocks against a volatile array of one-way hash values. Cattaneo et. al. implemented Transparent Cryptographic File System, TCFS, which validates file blocks using Hash-based Message Authentication Codes (HMACs) [14] that are embedded in the blocks [4]. However, both PFS and TCFS do not detect replays of old blocks. Mazieres and Shasha described a design of Secure Untrusted Data Repository, SUNDR, which has a storage organization similar to SFSRO, but stores the root hash in a secure location (presumably a client) [13]. Unlike GnatDb, the main focus of the design of the above file systems is the ease of integration of secrecy and tamper-detection to a file system. Consequently, their design is not optimized for low memory consumption and they do not support atomic and durable updates.

Blum et. al. considered the problem of protecting integrity of various data structures stored in an insecure memory using a Merkle tree with a root in secure memory [2]. This work provides a theoretical foundation for design of most of the systems that employ Merkle trees (unlike GnatDb). Schneier and Kelley described a mechanism for protecting logs against malicious corruption by computing a chain of one-way hashes and storing the tail of the chain in a secure, remote repository [17]. Verifying the log entries requires however recomputation of the entire chain and thus is unsuitable for database systems (such as GnatDb) that require an efficient random access to the data. Devanbu et. al. used a Merkle tree built on top of a relational database to validate result sets received from untrusted servers [8]. Similarly to SFSRO, but unlike GnatDb, the system is designed for read-only or read-mostly workloads.

PicoDBMS [3] is a database system designed to execute on resource-constrained smartcards. Unlike GnatDb, PicoDBMS does not provide secrecy and tamper-detection. On the other hand, it implements query processing. PicoDBMS aims to reduce the to-

tal database size by vertically decomposing relations into one column partitions to limit the repetition of column values. Although such storage model is appropriate for EEPROM memories that can be written one word at a time, it is less appropriate for stable storage that can be written only in large blocks (such as flash memories or hard disks), since update of a single tuple may result in writes to multiple blocks.

3 Architecture

To protect against malicious data corruption, GnatDb relies on the integrity of its security perimeter, which includes a processor, volatile memory, read-only memory and a one-way counter as shown in Figure 1. In particular, we assume that an attacker cannot modify the GnatDb executable in the read-only memory, cannot read or write the state written by GnatDb to the volatile memory and cannot read a secret value stored in the read-only memory. (The secret value is used to derive symmetric keys used by GnatDb to protect data written to the stable storage and its length should be sufficient to make it hard for an attacker to successfully guess its value.) We also assume that the one-way counter cannot be decremented, although an attacker might read its value or increment it (see [21] for an example of a hardware implementation of such a device). The volatile memory would be typically implemented using SRAM or DRAM and the read only memory using ROM or EEPROM (although GnatDb does not require write access to the read-only memory, it may be still desirable to be able to update the secret).

The physical security of the processor, volatile memory and read-only memory is often achieved by placing them on the same die [23], using tamper-resistant packaging and/or erasing the volatile memory after tamper detection [19]. Furthermore, the platform must either be able to protect the state of GnatDB in volatile memory and read-only memory (e.g., by using virtual memory) or it must load only trusted code [19].

On the other hand, we assume that the content of the stable storage can be read and arbitrarily modified by the attacker. It is difficult to ensure the physical security of large volumes of stable storage because it requires more time to erase than typical volatile memory, it can be analyzed offline or when the processor is powered off and its storage capacity is limited by the die size, if integrated to a single-chip system.

The architecture of GnatDb consists of three layers: Device, Secure Device and Store. Device, the bottom layer, implements a thin, common interface on top of the raw hardware. Secure Device implements the same interface as Device, but it also guarantees that all operations are secret and tamper-detecting. Store implements an interface that supports atomic and durable updates of sets of *records*, which are untyped, variable-sized sequences of bytes.

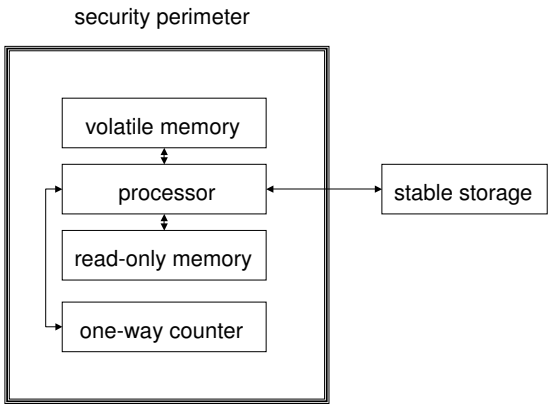

Figure 1: **Secure system model.**

3.1 Device

The Device interface consists of read, write, erase and flush operations. Device provides read and write access to fixed length *pages*. A page can be written atomically. One or more pages form a *block*, which is the unit of the erase operation. A write to a page that was not previously erased may raise an exception. Some storage devices may not require that pages are erased before being written (e.g., hard disks), in which case the erase operation is a no-op. The flush operation ensures that all written pages have reached a stable storage on devices that support volatile write caches (e.g., hard disks). Two examples of Device implemented on top of different types of flash memory are described in Section 6.1.

3.2 Secure Device

The Secure Device implements the same interface as Device, but it enforces the following security property:

Secrecy: Pages written by Secure Device can be read only by calling the read operation of Secure Device.

Tamper-detection: The read operation of Secure Device raises an exception if the value of the page being read is different from the value most recently written by the write or erase operations of Secure Device.

The Secure Device implements the security of GnatDb. Isolating all security functions to a single layer has several benefits: 1) the layer above (Store) can be implemented without considering security, 2) it is easier to verify the security of GnatDb when it is isolated in a single layer, 3) in appliances that do not require secrecy and tamper-detection, Secure Device can be transparently removed from GnatDb.

3.3 Store

The Store implements atomic and durable updates of sets of records. Each record must fit into a page. Records are persistently identified by *record id*'s. Records can be allocated, read, written and deallocated. The commit operation atomically and durably writes a sequence of records.

The `Store` interface consists of the following operations:

`RecordId allocate()`
: Allocates a new record id. The record id is reserved until the next commit.

`Buffer read( RecordId )`
: Returns the content of a record.

`void commit( <RecordId,Buffer>[] )`
: Atomically and durably writes a sequence of record ids and buffers. Records are deallocated by committing their record ids with an empty buffer.

4 Store Implementation

We first describe the implementation of Store in absence of the security considerations (by assuming the existence of a Secure Device implementation).

4.1 Storage Organization

GnatDb statically divides the space in the stable storage into two contiguous segments: the *index segment* and the *data segment*. The index segment contains metadata that maps record ids into pages in the data segment that contain the records. Both segments are log-structured [16], i.e., the updates are implemented by appending new versions at the tail of the log. GnatDb benefits from log-structured storage organization in several ways:

- Since records are not overwritten, implementing atomic updates is straightforward.
- All writes are compacted to a minimal number of blocks at the tail of the log. This is important on flash-memory-based Device implementations, where the cost of the erase operation dominates all I/O costs.
- If the log is written round-robin, it is possible to verify the integrity of the data in the log in a constant time without building a Merkle tree, as shown in Section 5.
- The erase operations are distributed uniformly across all blocks, which is important for flash-memories where each block can be erased only a limited number of times.
- Since records are not updated in place, traffic analysis of the accesses to the stable storage is harder. (It is difficult to link multiple updates to the same record.)
- Since records are never overwritten, it is straightforward to support variable-sized records.
- Compared to database systems using a log separate from the database, storage management in GnatDb is simpler in that it does not have to interpret two representations of a record (one in the

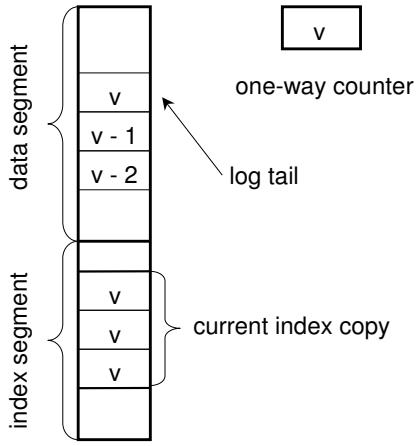

Figure 2: **Storage organization.**

The index segment contains multiple *index copies*. GnatDb writes a new copy of the index into the index segment each time it is updated in the store. We refer to the most recently written index copy (at the tail of the index segment log) as the *current index copy*. The frequency of these updates is discussed below. The index copy is a direct-mapped array, the location page id of the record with id *rid* is stored in the *rid*-th slot of the array. Since GnatDb is optimized for small databases, the index array can be compacted by allocating only a small number of bits for each slot. For example, the current implementation uses only eight bits to represent a page id in the index array, which limits the total database size to 255 pages. The index copy contains also contains metadata that is used during recovery, which includes the page id of the log tail, the length of the index, the number of deallocated record ids and the version number of the index copy.

The data segment consists of a sequence of *data pages*. Each data page contains a *page index* and a sequence of *record versions*. The page index maps record id's into locations within the page. The page index grows forward from the start of the page; record versions grow backward from the end of the page.

4.2 Record Id Allocation

Store allocates a new record id by either extending the index array (if there are no deallocated records) or by scanning the index array for the first deallocated record id. The allocated record ids do not persist across commits to reduce heap usage and to prevent

memory leakage for buggy applications that do not commit allocated record ids.

The operation requires in the worst case $|currentIndexCopy|_p$ Secure Device reads, where $|\cdot|_p$ denotes a size of an object in pages. It allocates at most one page buffer. (The buffer is allocated on the stack and does not persist across the Store calls because on many appliance platforms it is important that GnatDb occupies only a small amount of global memory. Consequently, allocating a larger number of buffers in main memory would not improve the performance of GnatDb.)

The index array scan could be sped up by linking all slots of deallocated record ids. This would, however, reduce the number of bits that encode valid page ids leading to a less compact index table.

4.3 Record Read

Store reads a record by first determining the id of the page containing the record, which is obtained by reading an appropriate slot of the index array. Subsequently, Store locates the record version using the page index.

The operation requires in the worst case two Secure Device reads and allocates one page buffer to read first an index copy page and then a data page.

4.4 Commit

During commit, Store accumulates the written records in a write page buffer. Once the buffer fills up, it is written to the data segment log tail. Subsequently, Store advances the log tail to the next page. After all records have been written to the data segment log, Store writes a new index copy at the tail of the index segment log. The new copy reflects the new locations of the records written during the commit. Completion of the index version write constitutes the commit point of the atomic update, i.e., any updates prior to this point are rolled back during recovery.

The number of writes could be optimized by organizing the index array into a hierarchy and writing only dirty pages. However, such design would lead to more complex index management (e.g., it would be harder to locate an index copy page given a record id or it would be harder to reclaim unused index segment pages). Since GnatDb is optimized only for databases with small numbers of records, such design would result in an unwarranted increase of code footprint due to the increased complexity of index array management.

Ignoring the cleaning overhead (which depends on the database utilization and is quantified in Section 6), the commit operation requires in the worst case $|(|records|_B + n_{records} \cdot recordOverhead)|_p + |currentIndexCopy|_p$ Secure Device writes and $|currentIndexCopy|_p$ Secure Device reads, where $|records|_B$ is the cumulative size of all committed records, $n_{records}$ is the number of committed records and *recordOverhead* is a per record-version storage overhead, which is the slot size in the page index (four bytes in the current implementation). This operation also requires three page buffers: one to read and write index copy pages, one to read data segment pages and one to accumulate the log tail writes. All three buffers are used simultaneously during cleaning. (Consequently, their number cannot be reduced.)

4.4.1 Cleaning

Before advancing the data segment log tail to a new block, Store must enforce the following invariant:

Clean Ahead: All live record versions in a block must be written to the stable storage before the block is erased.

A record version is *live* if the current index copy maps the record id to the page where the version is stored[1]. Clean Ahead implies that there must be at least one free block before erasing block b. In the design of GnatDb we chose to write the data segment log round-robin, i.e., the live record versions from block b are copied to block $b-1$ before any of the committed record versions can be written to block $b-1$. Writing the data segment log round-robin has several ramifications:

- Secure Device can easily verify the validity of each page if the pages are written sequentially to the data segment (see Section 5).
- Implementing cleaning is simple. There is also no need to thread non-contiguous blocks together to form the log. There is no need to maintain statistics about block utilization to select the best block for cleaning as is typically done in log-structured storage systems.
- Store may clean more blocks than necessary.

To avoid allocating read page buffers for the entire block (which would be prohibitive), each block is cleaned one page at a time by reading a page of block b and copying its live record versions to pages in block $b-1$. A new index copy is written once the buffers holding record ids and their new locations exceeds a preconfigured limit (128 bytes in the current implementation). This is important to keep the size of stack allocated data constant.

During a commit, Store may clean more than one block if the database utilization is high. Store prevents indefinite cleaning cycle by rasing an exception when it attempts to clean the same block twice.

[1]The definition is sound since a page can contain at most one version of the same record: Each page is written by at most one commit because pages must be erased between subsequent writes. If a commit attempts to write the same record more than once, the last write wins.

4.4.2 Recovery

During recovery, Store scans the index segment and finds the index copy with the highest version number (version numbers in headers of index copies form an increasing sequence). The index header also contains the page id of data segment log tail so that Store can resume writing the data segment log round-robin.

The storage organization could be simplified by eliminating the index segment and rebuilding the index copy in volatile memory at recovery by scanning the entire data segment. This, however, would lead to prohibitive memory storage and performance costs. Namely, the entire index copy would have to be buffered in main memory. In addition, recovery would be too slow on many appliances. (For example, reading a page from Secure Device takes approximately 3 ms on the platform described in Section 6.) Our design allows recovery to be performed quite frequently on appliances that are often powered off to improve battery life.

5 Secure Device Implementation

There are several possible attacks that can compromise either secrecy or tamper-detection:

- *Snooping.* An attacker reads the contents of a page directly from the raw hardware (secrecy violation).
- *Spoofing.* An attacker replaces a page with generated data (tamper-detection violation).
- *Splicing.* An attacker replaces a page with a duplicate of another valid page (tamper-detection violation).
- *Replay.* An attacker replaces a page with an older version of the same page (tamper-detection violation).

Secure Device prevents the first three types of attack using fairly common cryptographic techniques. Namely, it prevents the snooping attack by encrypting all pages with a symmetric secret key. It prevents the spoofing attack by including in each page a Message Authentication Code (MAC) [14] of its content computed with a secret key. It is computationally difficult for the attacker to produce a valid MAC without knowledge of the secret key. Finally, Secure Device prevents the splicing attack by computing the MAC over the id of the page in addition to the page content. The secret keys used for encryption and computing the MAC are derived from the secret value stored in the read-only memory (by e.g., by computing a one-way hash of the secret value and a deterministic salt).

To prevent the replay attack, Secure Device includes in each page a *version number*, which is protected by the MAC against forgery. Since both index and data segments are written round-robin, the expected version number of any page can be easily computed knowing the version number of the log tail. The one-way counter is incremented so that it holds the same value as the current log tail.

The data pages are assigned version numbers sequentially as they are written to the log. Similarly, all pages storing the current index copy are assigned the same version number, which is equal to the version number of the current data segment log. Finally, the one-way counter is incremented so that its value is the same as the version number of both log tails. The assignment of version numbers to pages is shown in Figure 2 in Section 4[2].

Recency of the index pages is established by matching their version numbers against the one-way counter value. (Store should never read older index copies, except in recovery. However, it is also possible to tolerate a limited discrepancy. For example, most hard drives enable write cache and therefore the OS cannot guarantee that the committed records have reached the stable storage. Consequently, the one-way counter may be ahead of the recovered current index copy version number.)

Recency of data pages is established by explicitly relying on the round-robin log write discipline. A data page written k steps before the log tail must have version number $c - k$, where c is the current value of the one-way counter (and also the version number of the log tail). More specifically, when reading data page pid_{read} from the data segment, Secure Device can verify that it is the most recently written version by computing an expected version number of the page as

$$c - [(pid_{tail} - pid_{read}) \ mod \ |dataSegment|_p]$$

and matching it against the version number found in the page. Secure Device raises an exception if a mismatch has been found.

Processor	EP7209 (ARM7)	74 MHz
DRAM	NEC 04265165G5	16 MB
NAND flash	Samsung SMFV008 SmartMedia card	8 MB
NOR flash	Intel TE28F320B3BA110	16 MB

Figure 3: **Hardware configuration.**

6 Experimental Evaluation

We ported GnatDb to a common embedded platform and used a benchmark that models a DRM workload

[2]To reduce the number of times the one-way counter is incremented, GnatDb allows the version numbers in the data and index segment log tails to diverge. The current index copy contains the version number of the data segment log tail.

	ARM7	x86
GnatDb basic (Bytes)	5624	6408
GnatDb + Secure Device (Bytes)	6872	7171
GnatDb + Secure Device + Rijndael (Bytes)	17740	34468

Figure 4: **Code footprint.**

to experimentally study GnatDb. We focused on quantifying the memory consumption as the primary objective and quantifying the performance as the secondary objective.

6.1 Setup

We ran all experiments on the Cirrus Logic evaluation board EP7209 [5]. The board hardware configuration (see Figure 3) is representative of higher end appliances. It includes a 32 bit RISC processor EP7209, which includes an ARM7 microprocessor, an MMU unit and 8 KB of on-chip cache. The board includes 16 MB of DRAM, which is common in most PDA's, but significantly more than in most single-chip systems such as smart-cards or secure processors.

The board includes several peripherials, including a SmartMedia NAND flash memory card, which is becoming a standard persistent bulk storage for appliances and a NOR flash memory. The SmartMedia card is read and written as 512 byte pages and erased in 8 KB blocks. The NOR flash memory is erased in 16 KB blocks and written and read four bytes at a time. To reduce the per-page overhead, the NOR-flash-based implementation of Device emulates 512 byte pages on top of the raw hardware.

We implemented the one-way counter using a reserved block of the NOR flash memory. To reduce the number of times the block needs to be erased, we represent the value of the counter k in the block as a sequence of $2^{17} - k$ ones and k zeros (an erased block consists of all ones). The counter has a wraparound of 2^{17} and needs to be erased once per a wraparound. Secure Device was configured to use Rijndael in CBC mode as a symmetric cipher and a Rijndael CBC-MAC [14].

Since most TPC benchmarks are too heavy-weight to model applications running in appliances, we designed a simple benchmark that models a typical DRM application. The database consists of n counters, with n being the scaling parameter of the benchmark. Each counter consists of a 20 byte static descriptor and a 4 byte value that is updated. The benchmark profile consists of a serial execution of 1000 transactions. Each transaction selects a random number of counters between one and five and sequentially reads the value of each counter, increments it by one and finally commits all new counter values.

6.2 Memory Consumption

We consider separately the static memory consumption, which includes primarily the code footprint of GnatDb library and the dynamic memory consumption, which includes stack and heap allocation. Firstly, the former is independent of the workload, but the later depends on the workload. Secondly, many appliances (such as smartcards or secure tokens) keep the executable code in ROM or EEPROM and use SRAM for the stack and the heap.

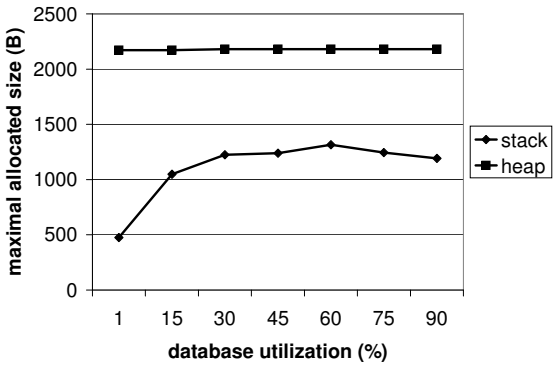

Figure 5: **Dynamic memory consumption.**

The static memory consumption depends primarily on the processor instruction set, the executable format and the compiler (we used Norcroft ARM C 4.9). For comparison, we also include the static memory consumption for x86 processor and Microsoft Visual C++ 6.0 compiler. The summary can be found in Figure 4. We show both the total basic code footprint that excludes Secure Device, which may not be needed in configurations that do not require protection against malicious database corruption, code footprint including Secure Device, but excluding the Rijndael implementation, which may be available on many secure processors in hardware or firmware, and finally the total code footprint. The Rijndael implementation[3] has a relatively large code footprint because it is optimized for speed (it generates a large number of tables). Smaller, size-optimized implementations that fit into 1.3 KB of memory are available [6].

The dynamic memory consumption is workload specific. We therefore use the DRM benchmark to study the consumption of stack and heap memory. The dynamic memory consumption includes an overhead that

[3] A free implementation available from Brian Gladman.

operation		avg time (μs)	min time (μs)	max time (μs)
NAND flash	read	303	293	996
	write	636	566	1461
	erase	2112	2104	2627
NOR flash	read	33	29	717
	write	3670	566	4572
	erase	339306	127998	373500
one-way counter	increment	32	27	35
Rijndael	encrypt	1144	1111	2758
	decrypt	1241	1210	2332
	MAC	1267	1232	1371
NAND flash Secure Device	read	3063	2973	4621
	write	3133	2996	4549
NOR flash Secure Device	read	2743	2640	4303
	write	6122	6033	7676
NAND flash Store	read	6277	6182	7600
NOR flash Store	read	5642	5551	6992

Figure 6: **Microbenchmark results.**

depends on the amount of cleaning. We therefore varied the database utilization by increasing n, the number of counters in the database, from 100 (2% database utilization) to 3000 (80% database utilization). The maximal stack and heap sizes for different database sizes can be found in Figure 5. The stack memory consumption levels out at approximately 2200 bytes, which is approximately 560 bytes over the size of the three page buffers that must be allocated for the commit operation. The heap consumption flattens at 1300 bytes. (The experiment also demonstrated the importance of limiting the size of relocation information that can be generated during cleaning, since without the check the heap size grew up to 5 KB.) Therefore, the total memory consumption of GnatDb, excluding the Rijndael implementation, does not exceed 11 KB.

6.3 Microbenchmarks

We ran a series of microbenchmarks to calibrate the performance of the stable storage devices, the basic cryptographic operations and the Store operations that have a fixed overhead. The results are summarized in Figure 6. All operations (except the one-way-counter increment) operate on 512 byte pages. The I/O times also include the time of transfer between the device and DRAM. The reported values are computed from a sample of 50,000 operations.

The one-way counter reset is identical to NOR flash erase and thus not reported in Figure 6. The Secure Device read and write times include the time to decrypt (encrypt) a page and to compute a MAC over a page, which accounts for approximately 40% to 95% of the total time. Since the Secure Device operations are crypto-bound, the performance of GnatDb could be improved by doing both encryption and MAC calculation in a single CBC pass [11]. The Secure Device erase is identical to the erase operation on the underlying Device and thus not reported in Figure 6. The Store read time includes the time of two reads from Secure Device. The Store commit time depends on the amount of cleaning and is the dominant cost of the transaction response time reported in 6.4.

6.4 Macrobenchmark

We used the DRM macrobenchmark described in Section 6.1 to evaluate overall performance of GnatDb. The transaction response time, which is dominated by the Store commit time, depends on the database size because the cleaning overhead grows with the database utilization as shown in Figure 8. The average response time of a transaction grows from 59 ms on a NAND flash memory (89 ms on a NOR flash memory) at 1% database utilization to 3 s on NAND flash memory (6 s on NOR flash memory) at 75 - 90% database utilization. The transaction response time remains at an acceptable level (less than 0.6 s) for utilization below 50%.

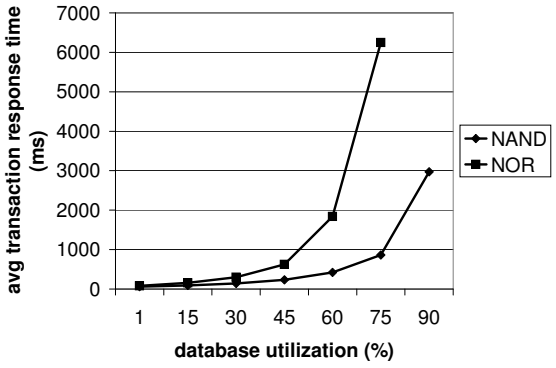

Figure 7: **Transaction response time.**

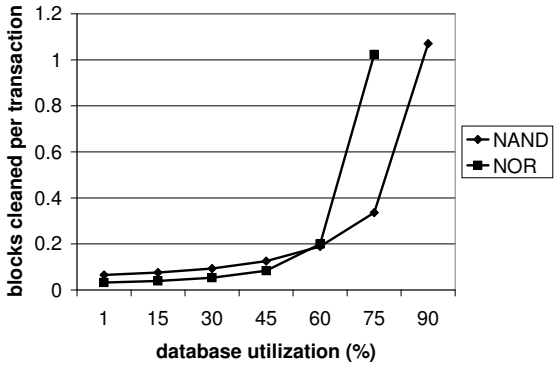

Figure 8: **Cleaning overhead.**

7 Lessons Learned

GnatDb has been designed by revising the design of TDB, which is a secure database system optimized for performance, to arrive with a secure database system that has a low memory consumption. In particular, we started with a database system with 250 KB code footprint with a goal to design a secure database systems with code footprint of less than 10 KB. Since research prototypes are not typically optimized for memory consumption, we believe that some of the lessons we have learned may be relevant, when generalized, to other research work.

The architecture of TDB consists of three layers. The bottom layer implements atomic updates of record sets, the next layer implements creation of incremental and full backups, the next layer implements object-oriented interface and the top layer implements iterator-based access to collections of objects. Our first (and the most obvious) cut was to concentrate only on the bottom layer, which had a 142 KB code footprint. For scalability, TDB implements location map organized as a hierarchy (which also embeds the Merkle tree). For good performance, the updates to the location map are performed lazily at checkpoints and the location map is reconstructed at recovery from the record updates found in the log. However, the implementation of the location map requires 39 KB of code. In GnatDb, on the other hand, we implemented location map as an array, which is sequentially written at each commit. Although clearly not scalable, the implementation fits into 1.8 KB.

TDB's cleaner maintains statistics about fixed size fragments of the log and selects for cleaning the fragments with the least utilization. The statistics are persistent and have to be checkpointed. This leads to considerable implementation complexity, since the checkpoints themselves could modify the statistics that are checkpointed. The cleaner of TDB is implemented in 46 KB of code. GnatDb, on the other hand, cleans both segments round-robin, which leads to a worse performance, but also to a simpler implementation with code footprint of less than 700 Bytes and also simplifies the replay attack protection to the degree that it is unnecessary to build a Merkle tree.

In our experience, it is hard to optimize code footprint as an afterthought. Code footprint optimization must be done throughout the design phase, when the benefits of a complex design must be carefully weighted against the corresponding code footprint increase. It is also important to avoid building generic modules with rich functionality that is not used by other modules. We found it useful to implement only the features that were clearly needed and add more features later if necessary (even at the cost of reimplementing the module). For example, the containers used in GnatDb do not support deletes, because the containers are always deallocated wholesale. Finally, although it is hard to quantify, the choice of an implementation language plays also an important role. For example, TDB is implemented in C++, while GnatDb in C (because there are no C++ compilers for many embedded platforms). The correlation between the sizes of source and assembly codes is weaker in C++ because the C++ compiler generates more assembly code automatically and in places that may not be obvious (e.g., copy constructors on arguments of function calls, destructors in scope exits). Consequently, a seemingly simple source code may frequently get compiled into a large assembly code.

References

[1] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the First ACM Conference on Computer and Communication Security*, November 1993. Fairfax, VA.

[2] M. Blum, W. Evans, P. Gemmel, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proceedings of the IEEE Conference on Foundations of Computer Science*, 1991. San Juan, Puerto Rico.

[3] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: scaling down database techniques for the smartcard. In *Proceedings of the 26th International Conference on Very Large Databases*, 2000. Cairo, Egypt.

[4] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001. Boston, MA.

[5] Cirrus Logic Inc., 4210 Industrial Dr., Austin, TX 78744. *EP7209 Development Kit*, September 1999.

[6] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. National Institute of Standards and Technology, July 2001. `csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf`.

[7] Dallas Semiconductor. *DS5002FP Secure Microprocessor Chip*, July 2001.

[8] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. In *Proceedings of the 14th IFIP 11.3 Working Conference in Database Security*, August 2000. Scoorl, The Netherlands.

[9] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000. San Diego, CA.

[10] N. Itoi. SC-CFS: Smartcard secured cryptographic file system. In *Proceedings of the 10th USENIX security Symposium*, August 2001. Washington, DC.

[11] C. Jutla. Encryption modes with almost free message integrity. IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, manuscript, (`eprint.iacr.org/2000/039`), August 2000.

[12] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000. San Diego, CA.

[13] D. Mazieres and D. Shasha. Don't trust your file server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001. Schloss Elmau, Germany.

[14] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996.

[15] N. Provos. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium*, August 2000. Denver, CO.

[16] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991. Pacific Grove, CA.

[17] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the USENIX Security Symposium*, 1998. San Antonio, TX.

[18] Dallas Semiconductor. Java-powered cryptographic iButton. www.ibutton.comibuttonsjava.html, July 2001.

[19] S. Smith, E. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Proceedings of the International Conference on Financial Cryptography*, 1998. Anguilla, British West Indies.

[20] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proceedings of the USENIX Annual Technical Conference*, 2001. Boston, MA.

[21] Infineon Technologies. Eurochip II - SLE 5536. Available at `www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_ov.jsp?oid=14702&cat_oid=-8233`, 2000.

[22] InterTrust Technologies. Rightschip. Available at `www.intertrust.com/main/products/rightschip-fs.html`, July 2001.

[23] J. Tual. MASSC: A generic architecture for multiapplication smart cards. *IEEE Micro*, 19, 1999.

[24] W. Wolf. *Computers as Components*. Morgan Kaufmann, 2001.

[25] E. Zadok, I. Babulescu, and A. Shender. CryptFS: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.