# A new Encryption and Hashing Scheme for the Security Architecture for Microprocessors

Jörg Platte, Raúl Durán Díaz, and Edwin Naroska

Institut für Roboterforschung, Abteilung Informationstechnik,
Universität Dortmund (Germany)
{joerg.platte,edwin.naroska}@udo.edu, raul.duran@uah.es

**Abstract.** In this paper we revisit *SAM*, a security architecture for microprocessors that provides memory encryption and memory verification using hash values, including a summary of its main features and an overview of other related architectures. We analyze the security of *SAM* architecture as originally proposed, pointing out some weaknesses in security and performance. To overcome them, we supply another hashing and protection schemes which strengthen the security and improve the performance of the first proposal. Finally, we present some experimental results comparing the old and new schemes.

## 1 Introduction

Protecting software is becoming more important for the future and therefore, efficient protection schemes are required. These schemes must provide a strong protection without requiring too many changes from the programmers point of view to be able to reuse existing code. Some processor extensions, like AEGIS [1] and *SAM* [2, 3], have been built to provide a secure execution environment for programs. Using these extensions, a program can be protected to prevent program code and data based attacks as well as runtime attacks. Hence, they are suitable to implement efficient copy protection schemes which cannot be removed or bypassed. Additionally, they can be used to protect program code and data disclosure by using encryption of memory contents and they must be able to protect against software based attacks (e. g., administrator access) and hardware based attacks (e. g., bus sniffing).

Protecting data or program disclosure is important in case of remote execution of programs. For example, in GRID computing, programs can be executed on many different computers spread all over the world and the submitter of these programs may not trust all remote systems. Using a security extension, the GRID can be used even for sensitive simulation data or secret algorithms.

This paper provides a security analysis of *SAM's* security functions and proposes modifications to its hashing and encryption algorithms. Using this modified scheme the security can be enhanced and the hashing performance increased compared with the old scheme.

Section 2 provides a brief overview about the *SAM* security architecture. Other security architectures are presented in section 3. Section 4 analyzes the

encryption and hashing functions and provides an optimized version. In sections 5 and 6 the simulation environment and the computed results are presented. Section 7 concludes this paper.

## 2 *SAM* Overview

*SAM* provides a secure execution environment for programs based on a standard processor design and a standard operating system. *SAM* aims at preventing tampering attempts as well as data and program disclosure.

The next paragraphs are providing a brief description of *SAM's* main attributes. A more detailed architectural description can be found in [2] and the design of the caches in [3].

The current *SAM* processor design is based on a SPARC V8 compatible CPU and was designed to be an optional extension. Hence, no secured bootstrapping or a persistent trusted operating system core is required to run *SAM* protected programs. Both protected and normal unprotected programs can be executed in a multitasking environment and small parts of the operating system are protected only while executing protected programs.

The processor core consists of an enhanced ALU supporting additional security instructions, an L1 data and instruction cache as well as an L2 cache. All data inside this core is trusted whereas all data outside is assumed to be untrusted. Hence, all data entering the L2 cache must be verified whereas all data written back to memory has to be protected against modifications and data disclosure. Data modifications are detectable by computing hashes for all protected cache lines. Prevention of data disclosure is achieved by transparent encryption of memory contents written to memory by the L2 cache. *SAM* uses the hashes both for memory protection and as a counter for a counter mode encryption algorithm to reduce the memory footprint. The hashing and encryption schemes are described in detail in section 4.

*SAM* uses a per process fixed virtual memory layout with two partly overlapping virtual address ranges. In the protected region all data is protected and verified by additional hash values and in the encrypted region, all data is additionally encrypted. All instructions located in protected memory regions (protected instructions) can access the decrypted memory contents in encrypted regions. Any other instruction can only access the encrypted data. Using virtual addresses simplifies paging of unused parts of the program or the hash tree.

*SAM's* caches have been suitably modified to provide these additional security functions. Additional security bits reflect the protection status for each cache line and dedicated security queues are used to hide additional latencies caused by verification and encryption/decryption. The memory write queues are computing hash values and encrypting cache lines to be written back to memory while the cache is able to process requests. A check queue contains all unverified cache lines and calculates hashes for these data in order to compare them with the ones in memory and detect memory based attacks.

All queues have a fixed size and therefore, stalls may occur when any of the queues is full. To prevent deadlocks, the L2-cache-bus arbiter monitors all queues and suppresses external cache accesses when the number of queue entries exceeds a given threshold. Figure 1 shows the relations between the L2 cache and the queues. In particular the check queue may exceed its threshold more likely, because each queue access may result in two additional queue entries, when requesting a hash results in a cache line replacement.

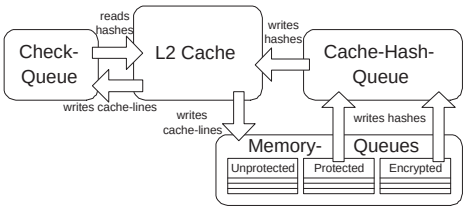


**Fig. 1.** Queue dependencies

While *SAM* ensures that all internal data like variables, constants and functions are protected, all external data read from files or sockets could be potentially untrusted. Hence, the programmer has to check all external data by using suitable cryptographic protocols.

## 3 Related Work

Using cryptography to protect algorithms and data in a tamper resistant environment is not a new approach. Secure co-processors have been proposed which provide a tamper-sensing and tamper-responding secure environment. These processors can be implemented on smart cards (for example, [4]) or as a co-processor shown by [5] in a PC (for example, the IBM PCIXCC [6]). These co-processors provide a secure environment. But they are limited in terms of processor speed and memory and often, programs must be significantly modified to be suitable for this kind of co-processors. Therefore, they do not provide an easy-to-use and expandable secure environment.

Another more related approach is the AEGIS architecture [1], the successor of the XOM architecture [7]. Like *SAM*, AEGIS provides transparent data and instruction encryption, decryption and verification of memory contents.

In AEGIS, a program consists of unencrypted, encrypted and protected parts and the architecture provides secure transitions between these parts. Variables and functions can be assigned to these regions at compile time by the programmer. Hence, the programmer needs a profound knowledge about possible attacks to not leak secure data.

In order to prevent software based attacks, AEGIS requires a special bootstrapping mechanism to load a security kernel that has access to the page table and other sensitive information. Memory contents are protected by hash trees based on their physical address. This requires free pages at subsequent physical addresses and prevents paging of these data without reencrypting them.

Each time a new program is started AEGIS first computes a hash over all secured program related data. This hash is then used in conjunction with processor and operating system hashes to decrypt the program. The initial hashing of a program is a time consuming and complex task, which has been implemented by executing internal microcode instead of a direct hardware implementation.

In addition to the hash values, AEGIS suggests the usage of 32 bit counters to encrypt data resulting in approx. 6 % additional memory consumption. This potentially gives rise to more misses during memory operations. Depending on the memory access this counter can overflow resulting in a time consuming reencryption of all program related data with a new key. Longer counters can prevent this for most programs, but they consume more memory.

## 4 Hashing Scheme and Encryption

In this section, we first revisit the hashing scheme and data encryption proposed in [2, 3], describing some weaknesses in security and performance. Then we propose some modifications on both the hashing and encryption schemes which improve the performance and strengthen the security.

### 4.1 Previous Hashing and Encryption Schemes

In general, a hash function $h$ with round function $f$ can be defined as follows:

$$H_0 = I_v, \quad H_i = f(x_i, H_{i-1}), \quad h(x) = H_m, \qquad i = 1, 2, 3, \ldots, m \tag{1}$$

where $I_v$ stands for initialization vector and $x_i$ are the $m$ fixed-length blocks which comprise the input $x$ (see, for example, [8, §9.4]).

**Hashing scheme** First we explain how to hash one cache line as per paper [2]. A cache line $C$ has 64 bytes (512 bits), further divided into four 16-byte (128-bit) blocks, for convenience. Each program is assigned a 128-bit key, $k_s$, at compile time.

A hash will be generated using AES as rounding function and a length of 128 bits for each block following the algorithm described in [9]. The value to be hashed is

$$C' = C \oplus (k_s || 0^{352} || V), \tag{2}$$

where $||$ represents concatenation of bits and $\oplus$ is the XOR operator, $k_s$ is the secret key, and $V$ is the 32-bit cache line virtual address. This operation ensures that no data can be copied to another virtual address and that any two identical data hash to different values when located at different virtual addresses. The key $k_s$ makes the hash value dependent on a secret value as well, and serves two purposes: firstly, it avoids possible exploitation of the hash value to extract information about the hashed contents, and secondly, several compilations of the same program will hash to different values, since $k_s$ is randomly chosen for each compilation. Let $C' = (X_1||X_2||X_3||X_4)$ be the value to hash, where each $X_i$ is a 128-bit wide block, and let

$$f(x, H) = E_x(H) \oplus H \tag{3}$$

be the rounding function, where $E_k(x)$ represents the application to $x$ of the AES function with key $k$. Then the hash is computed as follows:

$$H_0 = 2^{128} - 1, \; H_i = E_{X_i}(H_{i-1}) \oplus H_{i-1}, \; H = H_5 = E_{H_4}(H_4) \oplus H_4, \quad i = 1, 2, 3, 4$$

Observe that this computation adds an extra step at the end, when compared with the general hashing equations (1). The authors claim in [9] that this step is necessary since, otherwise, "an attacker could take a hash without knowing the corresponding file (i.e., the value to be hashed), and use it to generate the hash of a file which is the original file with an appended bitstring of arbitrary content." This hash generation needs five applications of the AES function.

However, data integrity cannot be guaranteed using only the previous scheme, because replay attacks are still possible. For this reason, Merkle trees (see [10]) are used, whereby each hash cache line, consisting of four hashes, is in its turn hashed and stored in a sort of tree manner. The uppermost level is called the root hash and consists only of one hash value, which protects the last four hash values, and it is stored permanently inside the processor. See [2, 3] for more details.

**Data encryption** Each cache line can be further encrypted using AES with the secret key $k_s$ in counter mode (see [11]). In this mode, an arbitrary value (the counter) is encrypted with the key $k_s$ and XORed with the plain data to encrypt, in this case, the cache line. The hash value described above is used as a counter. However, each cache line consists of four blocks, and so one hash value does not suffice as a counter, since this would mean reusing the same counter for four different blocks. To avoid this problem, it was suggested in [2] to XOR the hash value with four arbitrary (but architecturally fixed) bit patterns, $R_1, \ldots, R_4$, which could thus supply four different counter values.

**Performance evaluation** The obvious drawback of the described algorithm is that it is completely serial. Actually, the AES unit must used five times to compute the hash of one cache line. Therefore, the speed of one cache line hash computation can be only improved with a faster AES unit.

**Security analysis** The definition of $C'$ as per equation (2) presents the following undesirable property: two different cache lines, $C_1$, $C_2$ at virtual addresses $V_1$, and $V_2$, such that $C_1 \oplus V_1 = C_2 \oplus V_2$, will produce the same $C'$ and, hence, the same hash value. But, then, this means that if $C_1$ and $C_2$ are to be encrypted, they will use the same counter, which is completely unsafe.

Besides, the key $k_s$ serves two different purposes: it is used both to generate $C'$ and to encrypt data. It would be advisable to avoid this, just in case unexpected cryptographic primitive interactions may arise.

Last, it would be very interesting to get rid of the four arbitrary bit patterns $R_1, \ldots, R_4$, if possible.

### 4.2 New Proposal for Hashing and Encryption

The purpose in this section is to describe our new proposal for both the hashing and the encryption schemes, which improves the speed of operation, while even increasing the level of security.

**New hashing** The new hashing scheme has been suggested in [12, §2.4.4]. The idea is to replace the linear structure by a tree structure. This scheme, while not new, allows for a substantial speed-up in the evaluation of the hash function, which is now reduced to $O(\log m)$, where $m$ is the number of blocks to hash.

Suppose, as above, that $C$ is the cache line to hash. First of all we compute $C' = C \oplus r \oplus V$, where $r$ stands for a random value generated at compile time, and $V$ is the virtual address of the cache line. The reasons to use the values $r$ and $V$ are the same as in the old scheme. Note, however, that the computation of $C'$ does not preclude the problem that two different cache lines located at different virtual addresses could receive the same hash value. This problem will be addressed below.

We will use the same rounding function described in equation (3). Assuming again that $C' = (X_1||X_2||X_3||X_4)$, where each $X_i$ is a 128-bit wide block, the hash is computed as follows. First the following operations are performed in parallel:

$$H_1 = E_{X_1}(X_2) \oplus X_2, \qquad H_2 = E_{X_3}(X_4) \oplus X_4,$$

where $E_k(x)$ represents, as before, the application of the AES function to $x$ with key $k$. When the previous step is over, the following computation is carried out:

$$H = E_{H_1}(H_2) \oplus H_2,$$

Remark that in this case only two serial applications of AES are needed, versus five applications in the old scheme; this means a speed-up of roughly[1] 5/2. Last, observe that a final AES application is not needed (and thus can be saved), since the input has a fixed size and, therefore, the attack claimed by the authors in [9] cannot succeed in this particular case.

**New encryption** In this new scheme, AES in counter mode will be used, as before, but in a slightly different manner. First of all, each program will receive at compile time a base encryption key $k_b$. The encryption will be performed now at the block level, using a different key to encrypt each block; this encryption key will be derived from the virtual address of the block to be encrypted, using $k_b$ as a parameter. This "derived key" will be the actual encryption key for the block to encrypt.

More precisely, let $\mathcal{K}$ be the space of encryption keys, let $\mathcal{K}^\star$ be the space of base keys, and let $\mathcal{V}$ be the space of virtual addresses; then, given a block $X$, which belongs to cache line $C$ and is located at virtual address $V$, the encryption function is

$$E_{g_{k_b}(V)}(H(C)) \oplus X. \tag{4}$$

In this equation, $g\colon \mathcal{K}^\star \times \mathcal{V} \to \mathcal{K}$ is a suitable transformation function, such that, for each value of the parameter $k_b \in K^\star$, $g_{k_b}(V)$ supplies a usable AES encryption key. This function should satisfy the following property: for any $k_b$, $k'_b \in \mathcal{K}^\star$, then there do not exist $V, V' \in \mathcal{V}$, $V \neq V'$, satisfying $g_{k_b}(V) = g_{k'_b}(V')$. In practice, such function exists since $|\mathcal{V}| \ll |\mathcal{K}|$, but then, of course, the base key space is smaller than the original, namely, $|\mathcal{K}^\star| < |\mathcal{K}|$.

Initially, the requirement of hardware simplicity compels us to use a simple $g$, such as XORing $k_b$ and $V$. But, then, the system could be liable to a related-key attack, as described below.

[1] Some clock cycles are needed to initiate the second application of AES in the first step, so both operations are not strictly parallel.

**Security analysis** The encryption scheme is based on the use of the counter mode. As stated in [11], it is required that a unique counter is used for each plain text block that is ever encrypted under a given key. There is no particular indication on the counting values, as long as they satisfy the uniqueness requirement. This makes it possible for us to use the hash values as counters, since the design of $g$ and the protocol guarantee that they are unique for a given derived key. In fact, as it is easily checked, they only depend on the contents of the particular block to be encrypted.

Remark also that the new encryption method eliminates the need for the constants $R_1, \ldots, R_4$ described in section 4.1, since the encryption is now performed on a block basis. Besides, the random value $r$ has nothing to do with the encryption key $k_s$ of the old scheme, thus effectively decoupling both operations.

Finally, in order to keep an overall good performance, $g$ should be evaluable in a short time frame (for example, one clock cycle).

### 4.3 Revision of Some Common Attacks

We will subsequently revisit some possible attacks.

**Random attack** The opponent selects a random value and expects the change will remain undetected. If the hash function has the required random behavior, the probability of success is $1/2^n$, where $n$ stands for the number of bits in the hash. In our case, this attack is not feasible, since we are using 128 bit hashes.

**Birthday attack** In a group of at least 23 people, the probability that at least two people have a common birthday is greater than 1/2: this is called the *birthday paradox*. This fact inspires the so-called birthday attack, applicable when an adversary tries to generate a collision. Remember that the hashes are stored in a Merkle tree fashion, all of them in plain text except for the root hash, which is kept encrypted. The attacker is then forced to face the problem of finding a preimage for any of the hashes, since a collision is of no use. Therefore, this attack is not applicable. Moreover, a random preimage attack on a 128-bit hash code requires $2^{128}$, which can be considered unfeasible.

**Related-key attack** In this attack, the enemy is allowed to observe the operation of a cipher under different but mathematically related keys. In our case, cache lines are liable to hash to the same value under certain conditions, as stated above. Therefore, it is advisable that the transformation function $g$ used in equation (4) be selected so as to satisfy the necessary degree of randomness allowing the different "derived keys" to not disclose any mutual relationship.

## 5 Simulation Environment

This section briefly describes the simulation environment used to compute the results presented in section 6. The performance evaluation of different cache configurations is based on the SPEC benchmark suite. All benchmarks are executed in a virtual machine emulating a SPARC based computer with peripherals like

hard disk, framebuffer and keyboard. This virtual machine is based on the free system emulator QEMU [13]. QEMU achieves a good performance by translating all instructions of the guest system to native assembler instructions of the host system. Hence, all timing and memory access information are lost. Therefore, QEMU has been extended to add special monitoring instructions during the translation step to reveal this lost information. They are used to log instruction fetches, read and write data and I/O accesses by the CPU and memory access by simulated peripheral devices as well as context switches, interrupts, and the current clock cycle to a trace file.

| Cache property | Value |
|---|---|
| L1 placement | direct mapped |
| L1 line size | 32 bytes |
| L2 placement | LRU, 4-way-set |
| L2 line size | 64 bytes |

| Bus | Width | Divisor |
|---|---|---|
| L1 ↔ L2 cache | 128 bit | 2 |
| to memory | 64 bit | 5 |
| L2 cache ↔ Queues | 128 bit | 2 |
| to AES units | 128 bit | 2 |

**Table 1.** Cache properties

| Name | L1 size | L2 size | AES units | Check Queue entries |
|---|---|---|---|---|
| 8-256 | 8k | 256k | 5 | 3 |
| 16-1024 | 16k | 1024k | 5 | 3 |
| 32-2048 | 32k | 2048k | 5 | 3 |

**Table 2.** Cache configurations

This trace file is then used as an input file for the *SAM* cache simulator. It simulates an L1 data and instruction cache as well as the L2 cache with all security related queues as described in section 2 to compute the number of simulated clock cycles for these operations. Instruction and data access is passed to the corresponding L1 cache and external device access is simulated by occupying the memory bus. One limitation of using a trace file is the missing feedback from the simulator to QEMU.

The cache simulator is fully configurable in terms of cache sizes, bus widths, number of queue entries and their thresholds, clock divisors to simulate different clock rates for buses and components like the queues or the caches, memory latencies or hashing algorithms. The L1 cache runs always with maximum clock speed and all other components are clocked with divisors based on this clock rate. Table 1 lists the basic configuration used for all simulations.

For all simulations, all data between the virtual addresses `0x70000000` and `0xf0000000` has been encrypted. The hash tree starts at address `0x1aaaaab0`. A slightly modified Linux kernel has been used for the simulations. The kernel now starts to allocate memory for the heap starting at address `0x80000000` and all benchmarks are statically linked to the base address `0x70000000`.

For each simulated benchmark $2^{32}$ instructions have been written to a trace file after skipping the first $2^{32}$ instructions, approximately, which correspond basically to the initialization routines of each benchmark. Using this trace file a set of different cache configurations has been simulated to obtain the overall number of simulated cache clock cycles needed for all cache operations. This

set includes a configuration without security extensions which is further used as a reference for the speedup computation. The trace file does not contain any hash related data. During the simulation the cache simulator provides a random mapping of hash values to unused physical pages.

## 6 Simulation Results

Table 2 gives an overview about the configurations used for all simulations. The extension HT and HS are used for the tree and the sequential hashing algorithms, respectively.

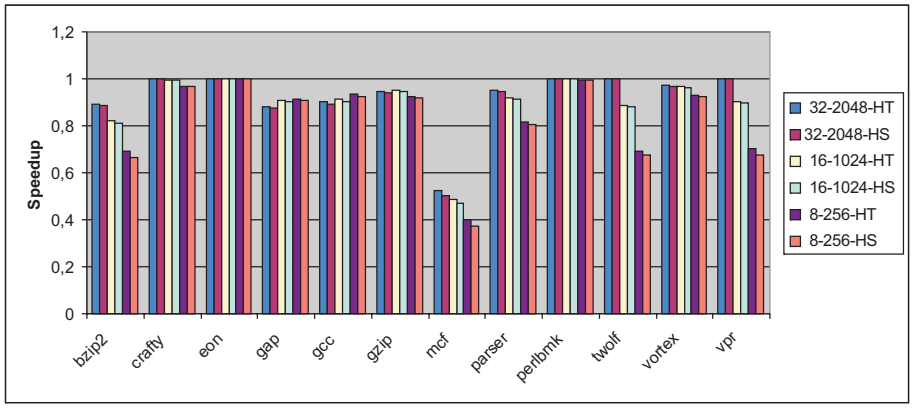


**Fig. 2.** Results for different cache sizes

Figure 2 compares both the sequential and the tree hashing algorithm for three different cache configurations. For nearly all configurations the speedup using the tree algorithm is higher than for the sequential algorithm. Also, the tree algorithm allows a more effective usage of the available AES units.

Using a larger cache does not always result in higher speedup as can be seen, for example, in the *gcc* benchmark[2]. Further investigation revealed that the number of cache line replacements (even for the cache configuration without security extensions) for those benchmarks is very similar for all three simulated cache configurations though the number of clock cycles is higher for the smaller caches. As a result, in this case even minor effects like different random mappings of hash values to physical pages (and therefore different sets) can distort this result.

[2] Remember, that the speedup for each benchmark has been computed in comparison with an equally configured cache without security extensions.

## 7 Conclusion

In this paper the cryptographic part of a processor security extension has been analyzed and optimized. The algorithm used for hash value generation has been optimized to provide a faster hardware implementation by parallelizing the algorithm and occupying more AES units at the same time. The presented benchmark results show that the presented new hashing algorithm further improves the good performance of the old scheme even without increasing the number of AES units.

The security analysis of the encryption scheme used by *SAM* does not prevent hash collisions for all cases. However, the proposed algorithm reduces the probability of collisions while slightly increasing the overall performance.

As a result, hash values can be used both for memory integrity verification and as a counter for a counter mode encryption scheme. Thus a significant saving in memory can be achieved compared to other architectures.

## References


1. Suh, G.E.: AEGIS: A Single-Chip Secure Processor. PhD thesis, Massachusetts Institute of Technology (2005)
2. Platte, J., Naroska, E.: A combined hardware and software architecture for secure computing. In: CF '05: Proceedings of the 2nd conference on Computing frontiers, New York, NY, USA, ACM Press (2005) 280–288
3. Platte, J., Naroska, E., Grundmann, K.: A cache design for a security architecture for microprocessors (SAM). In Grass, W., Sick, B., Waldschmidt, K., eds.: Lecture Notes in Computer Science. Volume 3894. (2006) 435 – 449
4. Sun Microsystems: Java card security white paper. `http://java.sun.com/products/javacard/JavaCardSecurityWhitePaper.pdf` (2001)
5. Yee, B.: Using secure coprocessors. PhD thesis, Carnegie Mellon University (1994)
6. Arnold, T.W., Van Doorn, L.P.: The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. IBM Journal of Research and Development **48** (2004) 475–487
7. Lie, D., Thekkath, C.A., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J.C., Horowitz, M.: Architectural support for copy and tamper resistant software (2000)
8. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL (1997)
9. Cohen, B., Laurie, B.: AES-hash. `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/aes-hash/aeshash.pdf` (2001)
10. Merkle, R.C.: Protocols for public key cryptosystems. In IEEE, ed.: IEEE Symposium on Security and Privacy, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, IEEE Computer Society Press (1980) 122–134
11. Dworkin, M.: Recommendation for Block Cipher Modes of Operation. Methods and Techniques. NIST. (2001)
12. Preneel, B.: Analysis and Design of Cryptographic Hash Functions. PhD thesis, Katholieke Universiteit Leuven (Belgium) (1993)
13. Bellard, F.: QEMU. `http://fabrice.bellard.free.fr/qemu` (2005)