# Transparent Code Authentication at the Processor Level

Ahmet O. Durahim, Erkay Savaş, Berk Sunar,
Thomas B. Pedersen, Övünç Kocabaş

September 12, 2008

**Abstract**

We present a lightweight authentication mechanism which verifies the authenticity of code and thereby addresses the virus and malicious code problems at the hardware level eliminating the need for trusted extensions in the operating system. The technique we propose tightly integrates the authentication mechanism into the processor core. The authentication latency is hidden behind the memory access latency, thereby allowing seamless on-the-fly authentication of instructions. In addition, the proposed authentication method supports seamless encryption of code (and static data). Consequently, while providing the software users with assurance for authenticity of programs executing on their hardware, the proposed technique also protects the software manufacturers' intellectual property through encryption. The performance analysis shows that, under mild assumptions, the presented technique introduces negligible overhead for even moderate cache sizes.

**Keywords:** Code authentication, stream cipher encryption, message authentication codes, universal hashing.

# 1 Introduction

The protection of computer systems from tampering and malicious code has become a major goal in recent years due to new business models that require a strong trust base in open personal computer systems. To cite a few, electronic commerce, electronic government, and online Banking require the handling of highly sensitive information. The lack of a root for trust in computing systems and the shear complexity of software systems, e.g. operating system (OS), prevents solid security schemes from being developed and deployed. Therefore, the establishment of trust in computing systems remains the major obstacle preventing widespread adoption of such key technologies.

To bring trust into computing systems is a sophisticated paradigm. The Trusted Platform Module (TPM) [41] was developed by the Trusted Computing Group as a standard for trust in computing. Despite the rapid advance in the standard achieved by strong backing of major device manufacturers and software providers, the techniques proposed in the standard provide only very general high level descriptions of the security and at this point do not consider performance related issues such as operational efficiency and architectural integration. For instance, a popular means for verifying an execution environment (*attestation*) is to

compute the hash of a program and/or data (state of a program). The computed hash value which is stored in TPM's Platform Configuration Registers (PCRs), are signed by the private key in the TPM to generate a *quote*. As shown in [32, 33], the performance of any operation involving TPM suffers significantly. Although initial or occasional use of the facilities of the TPM can be afforded, the heavy computational and communication (between processor and the TPM) burden eliminates the possibility of "on-the-fly" use of the TPM. Clearly, the solution is to integrate vital security mechanisms into the processor architecture; essentially embedding some of the TPM functionality (and possibly more that cannot be provided by the TPM) into the processor architecture. Some of the security mechanisms which have been proposed to embed in the processor are memory encryption, memory authentication, and code authentication.

Though memory and code authentication are closely related issues, there are important differences between the two. A memory authentication technique guarantees that memory, which may be under the control of an adversary, behaves as valid memory. Memory authentication, however, cannot detect if programs which run (legitimately) on the processor modifies data and/or code accidentally or maliciously. It is the role of code verification to guarantee that only "certified" programs can run on the processor, and that no program can, accidentally or otherwise, modify the code of itself or other programs. In this paper we propose a new, efficient method for code authentication.

Before and during the Trusted Computing initiative, less comprehensive schemes were proposed. In an earlier work, Chevallier-Mames et al. [9] proposed to employ a just-in-time delivery scheme of code which would eliminate the code tampering problem as the executed code would always be fresh, delivered just-in-time when needed and then verified and executed. This idea would address many of the security challenges we face today in embedded systems security. However, this new model brings new problems with it. The most critical one is performance. Most importantly, the latency introduced right before the execution will be unacceptable for most interactive users unless performed under the "one second" threshold. To overcome this obstacle, the authors advocate the use of fast secret-key algorithms over public-key schemes, and propose thread-level batch verification of instructions. Nevertheless, the reference does not give performance measures or a detailed architectural level description on the feasibility of the proposed scheme.

Similar security features have been studied in the computer architecture community as well. For instance, the Aegis architecture [39] aims to provide memory encryption and tamper resilience by instruction verification. The model proposes two different architectures differentiated by whether the OS is trusted or the trust is confined within the boundaries of the chip.

In [19] Gassend et al. propose a memory authentication scheme based on Merkle trees [34] to authenticate an arbitrarily large untrusted RAM memory. In [11] Clarke et al. propose a scheme which keeps the hash values of (logs of) all read and write operations. The memory can then be authenticated off-line by comparing the logs with the actual content of the memory. Furthermore, in [12] Clarke et al. propose a hybrid protocol with the aim of balancing the drawbacks and benefits of the previously mentioned two protocols. The overhead of this scheme tends to a constant as the number of instructions between critical instruction grows. We will later on discuss the connection of these works to the proposed authentication scheme.

In another work Yang et al. [43] focus on the related problem of achieving encrypted memory. The work provides more detail at the architectural level and achieves to hide much of the encryption latency behind the memory access cycles by utilizing a stream cipher like scheme. Despite the novelty of the design, the downside of this work is that there is no mechanism to prevent or detect tampering with the memory. In fact, if no additional steps are taken for integrity verification the proposed scheme is vulnerable to straightforward manipulations by an attacker. That is, the attacker may freely flip bits in encrypted memory which are translated to bit-flips in the plaintext (due to the stream cipher approach) with disastrous results. The paper proposes to use the hash tree based verification technique introduced earlier by Clarke et al. [11] to bring cryptographic integrity checks into their encryption technique. However, no treatment is given on whether this would be achievable without re-introducing the latency which the paper wanted to shift off the critical path in the first place. Another inconvenience of this approach is that it makes use of so-called sequence number codes which are used to eliminate the information leakage that occurs when the same memory location is modified and re-encrypted. Information leakage occurs due to re-encryption of the same memory block with essentially the same seed (or key) which means that the XOR-difference in the ciphertexts will reveal the XOR-difference of the plaintexts. Despite these shortcomings the work provides an innovative idea, i.e. hiding the encryption/decryption latency behind the access latency. The same idea was successfully applied to memory authentication and encryption in [42], that combines Galois Counter Mode of operation with Merkle's tree approach.

The main argument for moving memory and code verification away from the TPM is performance. The general purpose nature of the TPM makes it difficult to give efficient implementations. We claim that the best performance is obtained by treating memory and code verification separately, thus being able to optimize the methods for the different issues that arise in the two problems. Memory authentication needs to handle changes in data, but does not need to verify the origin of the data (it is written by the processor anyway). On the other hand, code authentication can use the fact that code is static, but has to ensure that it is the intented code.

In this paper we propose an efficient method for run-time authentication of code and static data. The proposed technique achieves to hide the bulk of the verification complexity behind the memory access latency. Our scheme relies on stream ciphers and universal families of hash functions. It is modular, so that the cipher and/or hash function can be easily substituted. We discuss how our code authentication scheme can be used in a number of scenarios with different security requirements. The security of the scheme is thoroughly analyzed. Finally, we present a performance analysis which shows that the presented technique introduces negligible overhead when caches of moderate sizes are employed.

# 2 Authentication in the Stored-Program Model

In the stored-program paradigm, the executable part of a program (i.e. instructions) along with data are stored in a single structure. This abstraction is realized physically in the *main memory* by modern day computers. The processor requests the instruction block to be executed by supplying its starting address to the memory. The memory provides the

instructions as requested. The memory also stores and supplies data, which can either be generated statically by the software publisher or dynamically by the program itself while in execution. Both instructions and data need to be authenticated for safe execution of programs in the processor.

The main memory is a place where many different programs and their associated data co-habit and therefore it is a particularly insecure place where instructions and data are subject to modifications by unauthorized parties and malicious programs. It is true that most computers employ some protection mechanisms that prevent programs from intervening address spaces of each other. However, the protection mechanism is enforced by the OS, which is overly complex and not necessarily a trusted piece of code. Hence, it is reasonable to assume that instructions and data can be modified during program execution.

The processor on the other hand, is commonly assumed to be a safe place once the instructions are brought from memory. The instructions and data are first put in an on-chip memory known as cache memory, whence the instructions and data are fetched for execution. The rationale behind believing that processor is a safe place for instructions and data is that no instruction can contaminate the instructions and data in the cache. Contaminated instructions and data are easily detected in our scheme since any update in the cache requires verification of the updated cache block before the execution of the instructions in the block[1].

In the program authentication model we propose, the instructions arrive in the cache along with their authentication tags, which are verified before they are executed. Instructions are transferred between the main memory and processor cache in groups, known as cache blocks or cache lines. While the instruction length varies from one byte to 17 bytes in contemporary processors, a typical length of cache block ranges from 8 to 512 bytes. In the proposed scheme, all the instructions in a cache block rather than a single instruction as suggested in [9] are used to generate an authentication tag. This approach has two advantages:

1. verification is faster, and

2. less overhead is incurred during the transfer of the tag from memory to the processor.

It is true that the proposed technique implies a small and modular modification to microprocessor core and that any modification to hardware is much harder than any modification to the software (more precisely to the OS where the most of the software protection mechanism is implemented). It is, all the same, certainly relevant and we strongly believe unavoidable, to explore these modifications since a software-only approach is proved to be inadequate for providing a proper security and trust. Furthermore, there is a strong perception that security, trust (and perhaps privacy) must be taken into account as design parameters in early stages of microprocessor development. Major microprocessor companies already introduced new technologies [15, 13, 3] that provides hardware support for security and trust. Similarly, many research groups have long been working on hardware modifications for providing security, trust, and privacy as an integral part of the microprocessors [44, 30, 12, 19, 11, 43, 9, 39].

[1]Other non-invasive attacks (e.g. fault induction) to contaminate the instructions in the cache can be thwarted using other means such as adversarial fault tolerant computing techniques (e.g. error detection); a vast topic which is beyond the scope of this paper.

In summary, the processor and main memory in the computer system are tied to each other through the instructions and data of a program and the former does not trust the latter. Restricting code verification within the boundaries of the CPU minimizes the trust base. This has significant consequences, as the OS no longer needs to be trusted. In fact, the operating system itself as well as the device drivers and the BIOS software, and virtually any piece of code can be authenticated *on-the-fly*.

The integrity of the instructions should be verified transparently with unnoticeable delay to the end user. For ease of use, convenience, and better scalability, signature schemes based on public key cryptography (PKC) seem to offer certain advantages. The key distribution problem is less of a concern, since the verification is realized using public key of the software owner or trusted software repository. Public keys, either stored in software or embedded in hardware, do not cause a security breach when they are compromised. But their authenticity must be validated using certificates, prior to verification. Nevertheless, there are two important factors that prohibit us from using PKC in code authentication: public-key signatures are too long, and public-key signature verification is slow. While long public keys can be tolerated up to a certain extent, high verification time is definitely unacceptable for on-the fly code authentication. Only symmetric cipher based authentication can provide both short authentication tags and fast (on-the-fly) verification of instructions.

In our scheme we utilize efficient *Message Authentication Codes* (MACs) [38, page 140] built using universal families of hash functions due to their high encryption throughput and short authentication tags. In addition to the performance benefits, MACs constructed using universal families of hash functions allow the designer to quantify the security level precisely, thus eliminating needs for performance degrading margins.

# 3 Alternative Protocols for Generation of Authentication Tags

None of the techniques for memory encryption/authentication proposed in [30, 43, 39, 11, 19, 12] addresses the issue of initial generation of the authentication tags and of cipherexts for instructions and static data. This concern is closely related to trust in computing since the user should be able to verify the authenticity of data and instructions before she uses them for the first time. The issue can be reduced to key distribution as the aforementioned techniques use a secret key for encryption and integrity check. Here we discuss several alternative scenarios along with the classical approach for key distribution (or more precisely tag generation) in the proposed scheme for software authentication. Unfortunately, there is not one single technique that can be used in all application scenarios where their requirements and constraints differ. Application developers and architects should choose a particular technique depending on these requirements and constraints.

Since the proposed code authentication technique utilizes a symmetric encryption to generate and verify the authentication tags, those parties which generate and verify the tags should be in possession of a secret key. However, distribution of the secret keys to involved parties is a major problem. In the classical setting, there are two parties in the authentication architecture: the *software producer* who owns the intellectual rights of the software and the

*software user* who wants to execute the software on his/her hardware in an authenticated way. The two parties share a secret key, $K$, that the software owner uses to generate the authentication tag, $T(P, K)$ for the program $P$. The secret key, $K$, may also be implemented in the user's processor by the hardware manufacturer to prevent the user from sharing the authenticated software with other users. The secret key can be unique either to a single user or to a group of users. Clearly, this will be less desirable to the user who has to trust the software owner. This technique can majorly be used in special-purpose processors such as those in game consoles and embedded processors deployed in automobiles where software producer is in complete knowledge of the hardware.

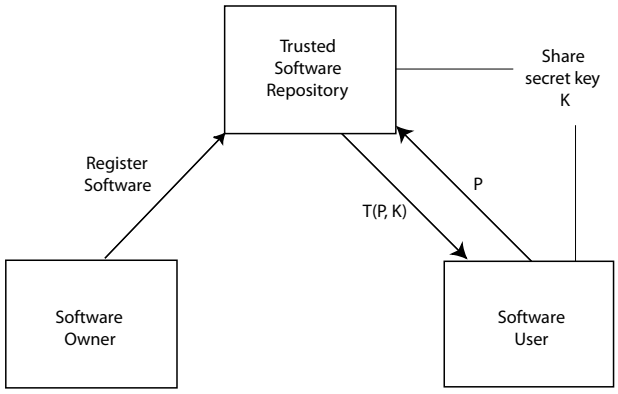


Figure 1: Authenticated Program Distribution Protocol based on a Trusted Software Repository

Another scenario illustrated in Figure 1 can help remedy the aforementioned concerns by involving the use of so-called *trusted software repository* (TSR). The TSR is responsible for generating authentication tags for the programs. The software owners register their software products with the trusted software repository, which inspects and stores the registered programs in its local database. Note that the repository does not have to store the entire program, but a representative of it, e.g. its hash in case the storage is of a concern. The repository shares secret keys with users and generates authentication tags for a specific program registered in the local database upon request by a user. The user first has to prove to the repository that it is indeed a legitimate holder of the program. The repository checks whether the program in question is identical to the one in the repository by comparing the hashes of the programs. If the first two steps are in order, the repository generates an authentication tag $T(P, K)$ for the program $P$ and sends it to the user. If the bandwidth between the user and the TSR is a concern, the TSR can only send, for instance, the hash of the all tags concatenated. The user generates all the tags for the program blocks herself and checks their integrity using the hash value she received from the TSR. This method can also be used in the other schemes described in this section and can help reduce the bandwidth requirements in case of software updates and fixes.

One concern with the centralized approach is that the TSR may quickly become communication and computation bottlenecks. This concern can easily be addressed using replication. The proposed protocol requires an infrastructure but otherwise does not require any changes

to the OS loader and no changes other than the authentication mechanism itself to the processor core. Although the technique provides a sound alternative for code authentication in general-purpose hardware it entails a trusted party whom the users must have complete confidence in; a situation some users may find discomforting due to privacy and trust concerns. On the other hand, the trusted party model and its variations [21] play a fundamental role in many cryptographic protocols. Thus, the TSR-based solution can be employed in code authentication whenever such a party exists.

The two techniques outlined above have one more shortcoming in common besides the aforementioned concerns: privacy. Zhang et al. in [44] pointed out that user privacy is an important issue even in the otherwise secure and trusted systems. Their solution is based on an architectural modification similar to our approach in this respect. One possible solution to privacy problem that will protect users against third parties tracking their activities is to utilize public key cryptography. The software owner signs its software using its private key and sends the software and the signature to the legitimate user who authenticates herself without revealing her true identity[2]. Upon downloading the software, the user, knowing the public key of the owner, verifies the software with the available trusted base (previously verified trusted — and authenticated — signature verification program or a TPM). During the verification process, the processor fetches the blocks of the software one by one starting from the first block. As it verifies the code using public key cryptography, the processor can generate authentication tags for blocks using the secret key; this is possible since tag generation and verification processes are identical. The proposed authentication infrastructure inside the processor which can verify authentication tags is also capable of generating tags without any modification. Consequently, neither the user reveals her identity nor the secret key leaves the trusted hardware. The techniques in [30] and [31] can be used for generating, protecting, and managing the secret key inside the processor. Although the tag generation by the user may incur high latency, it needs to be performed only once during the installation of the program and therefore the latency can be tolerated.

# 4 Details of the Code Authentication Procedure

In our scheme we make use of efficient MAC's to ensure the integrity of individual blocks of a program. The MAC is obtained by the application of a hash function picked randomly from a universal family of hash functions on the message, followed by the generation of the tag by encryption via a stream cipher.

The steps taken in the generation of authentication tags, illustrated in Figure 2, are summarized as follows:

- **Key Distribution** One of the critical phases of the authentication mechanism is the distribution of the authentication keys. For authentication, we need two keys: $K_U \in \{0,1\}^m$ which is used in the universal hash computation of the message block, and $K_T \in \{0,1\}^k$ which is used in the encryption mask generation through the stream

[2]The user can employ group or ring signatures or group keys as suggested in [44] to prove that she is one of the legitimate users. Any further discussion of anonymous authentication is beyond the scope of this work.

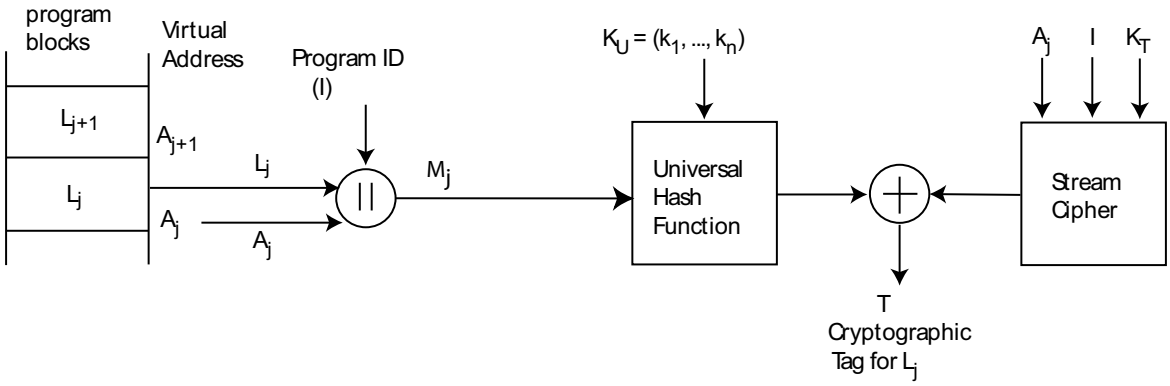


Figure 2: Cryptographic (authentication) tag generation

cipher. Here $k$ denotes the length of the key which will be used for the stream cipher. In the TSR-based scheme proposed in Section 3 the infrastructure eliminates the need for the user to obtain a key. The TSR creates the software authentication tags. The key is pre-built into the processor core. Hence, the TSR and the processor core share keys by pre-distribution. Furthermore, for tamper-resilience the key may be tied to the hardware by means of physically one way functions [31].

- **Software Preparation** The software (or rather a program since it is executable) $P$ is partitioned into program blocks[3], $L_j \in \{0,1\}^l$. The program has a unique identification number, $I \in \{0,1\}^i$ and each block has an address[4] specifying the placement of the block in the memory, $A_j \in \{0,1\}^a$. The message to be signed is $M_j := (L_j||I||A_j) \in \{0,1\}^m$, where $m = l + i + a$ and $||$ stands for the concatenation operation.

- **Digest Computation** Several efficient cryptographic hash functions have been proposed in the literature [8, 7, 29, 5, 26]. *Universal hash functions*, were first introduced by Carter and Wegman [8]. Roughly speaking, universal families of hash functions are collections of hash functions that map messages into short output strings such that the collision probability of any given pair of messages is the same as for a randomly chosen function. A universal family of hash functions can be used to build an unconditionally secure MAC. For this, the communicating parties share a secret and randomly chosen hash function from the universal family of hash functions, and a secret encryption key. A message is authenticated by hashing it with the shared secret hash function and then encrypting the resulting hash using the key. Carter and Wegman [8] showed that when the family of hash functions is strongly universal, i.e. a stronger version of universal families of hash functions where messages are mapped into their images in a pairwise independent manner, and the encryption is realized by a one-time pad, the adversary cannot forge the message with probability better than that obtained by choosing a

[3]A program block is mapped into a cache block which is the smallest amount of information transferred between the processor and memory.

[4]Virtual address of the block must be used since the physical address cannot be known at this stage.

random string for the MAC.

We use the hash function PR (NH-polynomial with reduction) proposed in [26] to generate message digests (or representatives) for our software blocks, $M_j$. The method for hashing, which is proven to be universal on $n$ equal-length strings (indicating that collisions cannot be forced to occur too often), is easy to use, and being based on binary polynomial operations in $GF(2^w)$ is easy to implement. Galois field $GF(2^w)$ also known as binary extension field is constructed using a binary irreducible polynomial of degree $w$, $p$. The elements of of $GF(2^w)$ are all the binary polynomials whose degrees are smaller than $w$. The arithmetic in $GF(2^w)$ are regular polynomial arithmetic with an additional reduction step by the irreducible polynomial (noted as $\pmod p$) whenever the degree of the result is larger than or equal to $w$. Since the carry propagation is not an issue in $GF(2^w)$ arithmetic, any constructions utilizing binary extension fields are preferred for its speed and small area.

The software block $M_j$ is written as a vector $M_j = (m_1, m_2, \cdots, m_n)$, where $m_l$'s are $w$-bit long and are considered as elements of $GF(2^w)$ for the hash computation. The digest is computed using a hash function chosen randomly from the family of hash functions PR as defined below.

**Definition 1** *[26] Given $M_j = (m_1, m_2, \cdots, m_n)$ and $K = (k_1, k_2, \cdots, k_n)$, where $m_l$ and $k_l \in GF(2^w)$, for any even $n \geq 2$, and a degree-$w$ polynomial $p$ irreducible over $GF(2^w)$, PR is defined as follows:*

$$D_j = \mathsf{PR}_K(M_j) = \sum_{l=1}^{n/2} (m_{2l-1} + k_{2l-1})(m_{2l} + k_{2l}) \pmod p \ .$$

In summary, as a result of message digest computation, a $w$ bit message digest, $D_j$, is obtained for an $m$ bit program block, which is the concatenation of instruction block, block address, and program identification number in encrypted form and $w < m$

- **Cryptographic (Authentication) Tag Computation:** The tag is computed as $T_j = D_j \oplus R_T$ where the pseudo-random pad $R_T$ is the block output of the utilized stream cipher, i.e. $R_T = \mathsf{SC}(K_T, A, I)$. The lengths of $T_j$, $D_j$ and $R_T$ are naturally identical. The generated cryptographic tag is stored in memory as explained in Section 6.

Cryptographic tag generation is performed off-line and therefore does not constitute a performance bottleneck. The software validation process, explained below, is performed on-the-fly while the instructions are fetched from the memory, and thus naturally raises concerns of performance degradation. However, we show that almost all computation during software validation can be done in parallel to the memory access operation.

## 4.1 Software Validation

The outline of software validation is illustrated in Figure 3. When a memory request for an instruction block goes out, the memory access latency is used to generate the pseudo-random

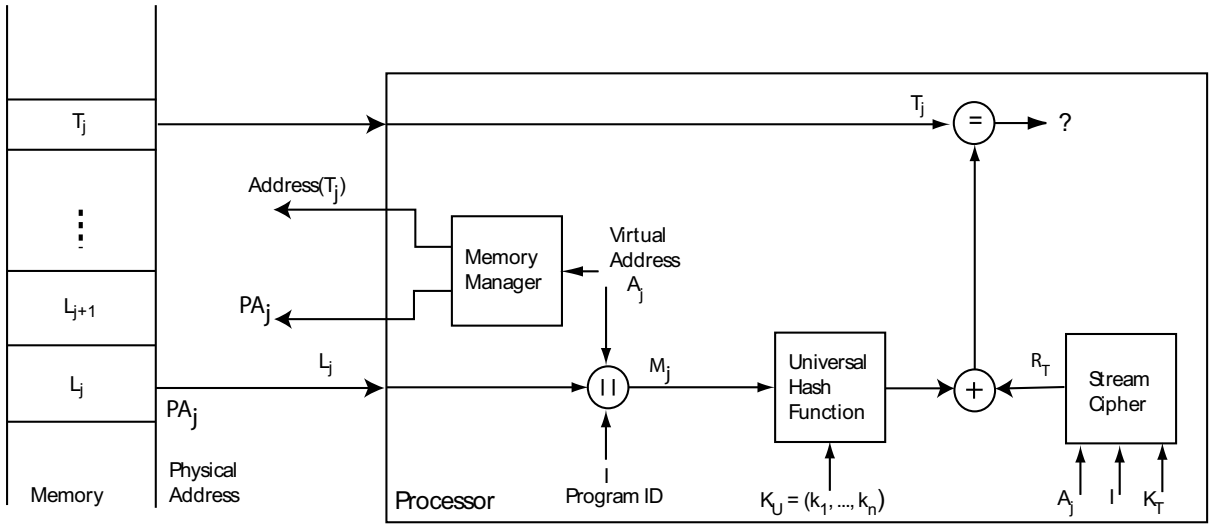


Figure 3: Validation of software blocks

pad $R_T$ with the stream cipher $\mathsf{SC}(K_T, A, I)$. The existence of extremely fast implementations of stream and block ciphers justifies the assumption that sufficient pseudo-random bits can be generated for an instruction block [36, 27, 18, 2] in the time it takes to perform a memory access, i.e. the memory access latency. The resulting pad $R_T$ is accumulated in a register within the processor.

As instructions are retrieved from the memory into the CPU, they are incrementally hashed using the fast universal family of hash functions. Each incremental hash operation consists of only two additions and one multiplication in $GF(2^w)$ which can be performed extremely fast. After the last block is hashed, the accumulated tag value is computed as $T_j = \mathsf{PR}_{K_U}(M_j) \oplus R_T$. The tag $T_j$ computed in the authentication unit must be identical to the tag that are fetched from the memory, in which case the verification is accomplished. As long as the latency of an incremental hash computation operation can be hidden behind the memory access latency, the verification operation will run concurrently. The result of the verification as well as the instructions will be ready right after the last block is brought in from memory and incrementally hashed. Hence, the overhead introduced by this step, is in the retrieval of the authentication tag from memory and in the latency of the last incremental hash and XOR computations. The latter incurs insignificant overhead, i.e. 1 clock cycle, compared to the overhead due to the former[5]. The overhead introduced due to both factors is studied in Section 7.

When an instruction block along with its authentication tag, $(M_j, T_j)$ is received, the authentication circuit that is built into the processor first computes the digest $\mathsf{PR}_{K_U}(M_j)$ as defined above, where $M_j := (L_j||I||A_j)$ and $L_j$, $I$, and $A_j$ are program block, program ID, and block address, respectively. The circuit checks whether the tag verifies and decides to execute (or not) accordingly, as shown in Figure 3.

[5] "Early-start" and "critical-word-first" techniques cannot be applied in the proposed scheme since the processor waits for the entire block to execute the requested instruction, which may arrive earlier.

### 4.2 Encryption of Software Blocks

The proposed scheme provides only integrity for program blocks in order to prevent unauthorized pieces of software from executing in the processor. However, we can easily enhance our scheme with encryption to protect the IP of software manufacturer. The executable code can be kept in encrypted form in memory. Using the stream cipher approach in [43], the computation time for encryption and decryption operations can be hidden behind the memory access latency. The only concern is whether a sufficient amount of pseudo-random pad can be generated during the memory access. The number of pseudo-random bits required for encryption depends on the size of the instruction block. Assuming we partition the software using the cache block size, the number of bits required for encryption/decryption can be as many as 512 bytes. The number of bits required for the authentication procedure explained above, on the other hand, are only 16 bytes (128 bits) for an acceptable level of security. Fast stream and block cipher implementations [36, 27, 18, 2] justify the assumption that the required number of pseudo-random bits can be generated during the memory access operation.

## 5 Security Analysis

We follow the strategy developed by Krawczyk in [29]. That is, given a message $M$ authenticated by the tag $t = h(M) \oplus r$ where $h$ is randomly selected from some family of hash functions and $r$ denotes a random string, the adversary should not be able to find $M'$ ($\neq M$) and $t'$ such that $t' = h(M') \oplus r$ with non-negligible probability[6]. Here it is assumed that the adversary knows the description of the family of hash functions but not the chosen $h$ or $r$ values. In this scenario Krawczyk defines a family of hash functions to be $\epsilon$-AXU (almost-XOR-Universal)[7] if it resists such an attack with probability larger than $\epsilon$. The following theorem quoted from the same reference establishes the necessary and sufficient condition to obtain such a MAC.

**Theorem 1** *A necessary and sufficient condition for a family $H$ of hash functions to be $\epsilon$-AXU is that for all $M_1 \neq M_2$ and $c$ of $w$-bit constant,*

$$Pr_h[h(M_1) \oplus h(M_2) = c] = \epsilon \ .$$

In the same reference, Krawczyk introduces an LFSR-based universal family of hash functions which is proved to be $\epsilon$-AXU. In practice, any $\epsilon$-AXU family of hash functions may be used in the code authentication scheme proposed in this paper, as long as it is possible to build a circuit with reasonable footprint which will hash a block of code, in less time than the memory access latency. Here we use the PR universal family of hash functions, since reference [26] gives detailed hardware implementation results which gives hard evidence that this performance condition will be met by this family of hash functions.

Although not shown in the original reference, the universal family of hash functions PR may be easily shown to be $\epsilon$-AXU (cf. Appendix).

[6]The symbol $\oplus$ denotes the parallel bitwise exclusive-or operation.

[7]The reference by Krawczyk uses instead the terminology $\epsilon$-otp secure. We prefer to use the equivalent but more common terminology, i.e. $\epsilon$-AXU in the text.

**Theorem 2** *For any even $n \geq 2$ and $w \geq 1$, $\mathsf{PR}[n, w]$ is $\epsilon$-AXU on $n$ equal-length strings, for $\epsilon = 2^{-w}$.*

As a direct consequence of Theorems 1 and 2 the family of hash functions $\mathsf{PR}$ is $\epsilon$-AXU. This means that the MAC obtained by computing $T = \mathsf{PR}_K(C) \oplus R_T$ is $\epsilon$-AXU secure with $\epsilon = 2^{-w}$ when $R_T$ is randomly chosen and used only once. Alternatively, if $R_T$ is generated by a pseudo-random number generator (or stream cipher), the security of the MAC rests on the security of the pseudo-random number generator.

Finally, we would like to note that there is an implicit benefit of using universal hash functions in this application. Since the security level can be quantified through the $\epsilon$ value, the length of the authentication tag can be precisely optimized to be minimal for a required security level. For instance, for practical applications $\epsilon = 2^{-80}$, or in other words a 80-bit authentication tag per cache line should suffice to provide an acceptable level of security. However, we use 128-bit authentication tag per cache line in order to demonstrate the efficiency and feasibility of our scheme for even a higher level of security.

# 6 Processor and Memory Organization

The processor and memory organizations need to incorporate small changes in order to support the proposed model for authenticated code execution. Figure 4, depicts a generic memory organization adopted from common MIPS processors, where $GP$ and $PC$ stands for global pointer — for easy access to static and global data — and program counter, respectively. For easy access, the authentication tags are placed next to the data segment. The order of the tags is the same as the instruction order in the memory. The address of the tag of the first instruction is kept in a special purpose register, named here as *tag pointer*, in short $TP$. Inside the processor architecture, as depicted in Figure 5, in addition to $TP$, a small memory (named as *authentication cache*) is kept for storing authentication tags, and an authentication unit is included for validation of instructions. The authentication cache contains authentication tags of instruction blocks, and is assumed to be of the same size as one cache line. The authentication cache uses the same working principle as the instruction/data cache. When an instruction is to be authenticated, its tag is searched in the authentication cache first; in case it is not found in the authentication cache, the current content is evicted and the required authentication tag block is fetched from the memory. We assume that a memory block (which resides in one cache line) is the smallest amount of information transferred between the processor and the main memory, independent of the type of information (i.e. instruction, data, or authentication tag). Therefore, a program block is identical to a block of instruction cache.

The authentication unit inside the processor, which incorporate a stream cipher $\mathsf{SC}$ and other necessary circuit, computes the authentication tag, as the instructions arrive in the instruction cache in a cumulative fashion. The computed tag for the instruction block is then compared against the tag in the authentication cache. If they match, all the instructions in the cache line are considered as authentic and ready for safe execution.

If the authentication fails, the action taken by hardware or software (e.g. the OS) depends on the application. The safest method is to terminate the program execution entirely

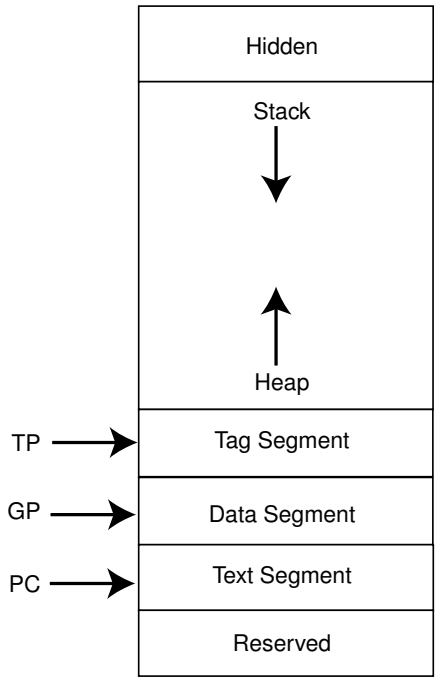


Figure 4: Memory Organization

in security-related applications. If the program has the privilege of accessing sensitive information (e.g. secret keys, private information), it is best to prevent the execution of any untrusted instruction since the damage can be irreversible otherwise. On the other hand, there may be some other applications where we can just discard the results in case the program that generates the results fails to authenticate in a certain instruction. In this case, the program does not have to be terminated; but a flag may be set to indicate an authentication failure.

While requested instructions are accessed through PC (program counter), the corresponding authentication tags are accessed using both PC and TP. TP is initially set to the address of tag for the first instruction block. As the PC is updated (either incrementing it by 4 or branch/jump target address), TP is updated using the formula

$$TP := TP + \left\lfloor \frac{PC}{n_I \times n_T} \right\rfloor ,$$

where $n_I$ and $n_T$ are the number of instructions and number of authentication tags in a cache line, respectively.

# 7 Performance Analysis

In this section we demonstrate the efficiency and practicality of our scheme employing the standard clock-cycle per instruction (CPI) metric as a basis for our performance analysis with 128-bit authentication tags per cache line for a good security level. In this section,

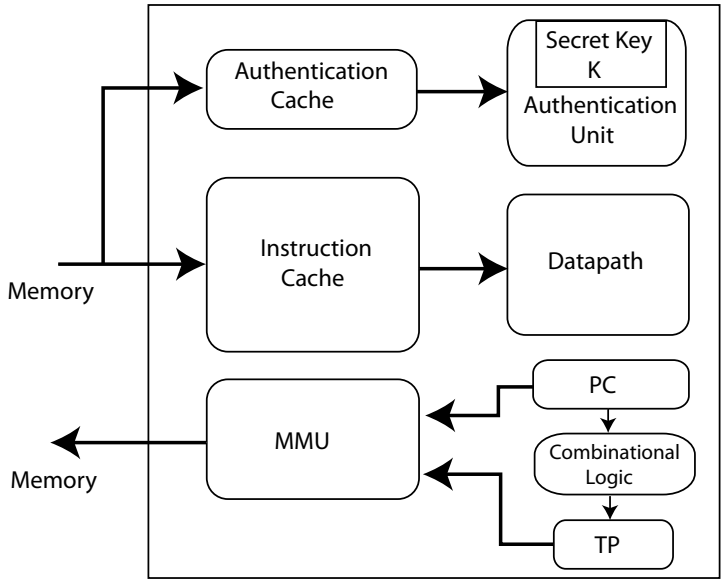


Figure 5: Processor Architecture

we compare the performance of the authenticated architecture w.r.t. a base model which simply stands for a regular processor architecture without the proposed code authentication feature. As described earlier, although the proposed design applies on virtually any processor architecture, we base our analysis on a standard RISC architecture with five pipeline stages: IF (inst. fetch), ID (inst. decoding), EX (execute), MEM (memory access), WB (write back).

The performance parameters related to the software profile are given in Table 1.

Table 1: Performance parameters related to software profile

| Parameter | Definition |
|---|---|
| $f_L$ | Frequency of load instructions |
| $f_D$ | Frequency of data dependencies caused by load instructions |
| $f_R$ | Frequency of data dependencies caused by load instructions and resolved by reordering |
| $f_B$ | Frequency of branch instructions |
| $f_J$ | Frequency of jump (unconditional branch) instructions |
| $f_{\text{Taken}}$ | Percentage of branch instructions that are taken |

Similarly, architectural parameters determined by the organization of the processor and memory system are enumerated in Table 2. $c_{AR}$ stands for the elapsed time between when the first word of data appears at the memory output and the address is sent to memory. This parameter primarily depends on the memory technology and the size of the memory. $c_T$ represents the time required to transfer an entire block of instructions from the memory

output through the bus into the instruction cache after address resolution. This parameter depends on the size of the instruction block and the bus width.

Table 2: Architectural parameters

| Parameter | Definition |
|---|---|
| $c_{AR}$ | Time spent (in number of clock cycles) for address resolution |
| $c_T$ | Time required to transfer an instruction block from the memory to cache |
| $n_I$ | Number of instructions in a cache line |
| $n_T$ | Number of tags in a cache line |
| $f_{MP}$ | Percentage of branch mispredictions |

## The Base Model

The equation modeling the $CPI$ in the base model is given as

$$CPI = 1 + C_L + C_B + MR_I \times MP_I + MR_D \times MP_D \ .$$

The term $C_L$ represents the overhead contributed to the CPI due to data dependency associated with load instructions, i.e., when a data load instruction is followed by an instruction that uses the result of the load instruction causes a delay in many common processor architectures. Here we assume that in the five-stage pipelined RISC architecture the delay is one clock cycle. This assumption is based on the fact that, in the pipeline that employs data forwarding from the MEM stage to the EX stage, it is sufficient that the next instruction stalls for one clock cycle in the ID stage. The majority of such dependencies are resolved by reordering of the instruction by the compiler. Still we can formulate the effect of data dependency related delays in the analysis as

$$C_L = f_L \times f_D \times (1 - f_R) \ .$$

The term $C_B$ represents the overhead contributed to the CPI from branch mispredictions. We assume the RISC architecture includes a simple branch prediction scheme of *assume-not-taken*, where the branch outcome is resolved in the ID stage. Therefore, one clock cycle is lost on misprediction.

$$C_B = f_B \times f_{MP}$$

The product $MR_I \times MP_I$ stands for the latency incurred in instruction cache misses. The term $MR_I$ stands for the instruction cache miss rate and depends on the size of the instruction cache as well as on the statistics of the branching distance of the software profile. The miss penalty, $MP_I$, represents the number of clock cycles needed to bring a block of instruction from the main memory to the instruction cache. Similarly, $MR_D \times MP_D$ stands for the latency incurred in data cache misses. $MP_I$ (or $MP_D$) has two components: address resolution and the transfer of the instructions through the bus into the cache. Hence, $MP_I = c_{AR} + c_T$. The most explicit form for the CPI is given as follows

$$CPI = 1 + f_L \times f_D \times (1 - f_R) + f_B \times f_{MP} + (MR_I + MR_D) \times (c_{AR} + c_T) \ .$$

where we assume that instruction and data caches have the same configurations.

## The Authenticated Processor Model

The CPI performance of the processor architecture that also includes an authentication unit is modeled as

$$CPI_{Auth} = CPI + C_{Auth} \ .$$

The overhead caused by the authentication is broken down as follow

$$C_{Auth} = 1 \times MR_I + MR_I \times MP_I \times \text{factor}$$

where factor represents the rate of authentication (cache) misses while the first term accounts for the overhead due to one extra clock cycle to compute the final step in the cryptographic tag calculation. Authentication misses are closely related to instruction flow during program execution and minimized for sequential execution. Control flow instructions (e.g. branch and jump) occasionally change instruction flow which results in higher miss rates for both instruction and authentication caches. Since authentication cache can contain more than one tag, it sometimes does not miss even if there is an instruction cache miss[8]. The term factor represents the fraction of instruction misses that also result in an authentication cache miss.[9]

The range of instructions, whose tags are in the same authentication block, is hereafter called *window of instructions*. The number of instructions in a window can be calculated using the simple formula $n_T \times n_I$. We can categorize the authentications misses depending on their cause, therefore obtain a more refined formula for factor as follows:

$$\text{factor} = \text{factor}_{SE} + \text{factor}_B + \text{factor}_J \ .$$

The term $\text{factor}_{SE}$ represents the number of authentication misses during sequential execution of the instructions. Namely, the first instruction in a window always result in an authentication miss even if there are no branch or jump instructions. The formula to calculate $\text{factor}_{SE}$ is derived by discarding the jump and taken branch instructions:

$$\text{factor}_{SE} = \frac{1 - f_B \times f_{\text{Taken}} - f_J}{n_T \times n_I} \ .$$

A branch instruction, when taken, causes an authentication cache miss when the target instruction is beyond the current window of instructions. Therefore, not all taken branches result in an authentication miss since the target instruction can be in the current window of instructions. We use the term *branch distance (BD)* to denote the difference in the address of the current instruction and the address of the target instruction. Similarly, branch offset $b$ is the number of bits that encode the distance in the branch instruction, i.e. $b = \lceil \log_2(BD) \rceil$. The statistics given in [22] shows that the majority of the branches are to relatively close locations, where the branch offset can be captured with only 7-8 bits. Consequently, whether a branch instruction leads to an authentication cache miss depends on both the branch offset

[8] We assume that the smallest cache block is 128 bit.

[9] The case that instruction cash hits while authentication cache misses does not entail fetching the authentication tag from memory. Instructions found in the cache must have been already authenticated and hence trusted.

and the position of the branch instruction in the instruction window. For instance, a forward-taken branch, which is placed near the end of the window, will cause an authentication cache miss with a very high probability.

The probability that a branch instruction causes an authentication miss increases in proportion to the increase of the branch offset. When the branch offset is for example $b = 2$, the branch displacement will be at most 4, and therefore the last four instructions[10] may cause an authentication cache miss if they are forward-taken branches. Using all branch offset values in *PC-relative* addressing common in RISC processors, i.e. $n = 1 \ldots 15$, we can derive a formula that gives the probability that an instruction causes an authentication cache miss as follows:

$$\text{factor}_B = \sum_{b=1}^{15} f_B \times f_{\text{Taken}} \times \frac{\min(2^b, n_I \times n_T)}{n_I \times n_T} \times f_b(b),$$

where $f_b(b)$ represents the percentage of all branches whose branch offset is $b$-bits[11].

Unconditional branch instructions, including function calls, are usually referred to as *jump* instructions and therefore are always assumed to cause authentication cache misses. Therefore, we have $\text{factor}_J = f_J$.

The MIPS dynamic instruction mix for five SPECint2000 benchmark programs (i.e. gap, gcc, gzip, mcf, perl) reported in [22, page 148] shows that branch and jump instructions makeup 12% and 1% of instructions, respectively. In other words, $f_B = 12\%$ and $f_J = 1\%$[12]. For the same benchmarks, the ratio of conditional branches, $f_b(b)$, whose branch offset is $b$-bit is also reported in [22, page 173]. The frequency of taken branches $f_{\text{Taken}}$ for the SPEC2000 programs is reported as 66% in [22, page 232]. The instruction miss rate varies with cache and block sizes as well as the workload. The miss penalty depends on block size and width of the bus between the processor and memory. Using the miss rates in [22, page 390] for five SPECint2000 programs and miss penalty values in [22, page 414] for a fixed block size of 64 bytes, we can calculate the CPI overhead due to instruction authentication. In Figure 6, we illustrate overheads to CPI due to instruction authentication, instruction cache and data cache misses for different cache sizes (and therefore different instruction and data miss rates). Figure 6.a distinguishes CPI overheads due to these three factors using different line styles. Figure 6.b illustrates total CPI overhead due to instruction authentication, instruction and data cache misses, with different shades representing the overheads due to different factors. Note that, the CPI overhead due to instruction authentication is hardly noticeable (darkly shaded area on top) when compared the CPI overheads due to instruction and data cache misses (lightly shaded areas at the bottom). Thus, we compared the CPI overheads due to instruction authentication and instruction cache misses in Figure 6.c, where the effect of instruction authentication is now slightly more noticeable. The proposed model benefits from large cache blocks, and hence incurs higher overheads when the block size is small. However, the overhead due to instruction authentication is still acceptable for even the smallest block sizes common in embedded processors. The results in Figure 7 demonstrate

[10]For a conservative estimate we take the maximum amount of displacement for a given number of offset bits.

[11]This unified formula incorporates the effects of both forward and backward branches

[12]For a conservative estimate we do not use SPECfp2000 benchmarks whose statistics features much lower frequencies of branch instructions

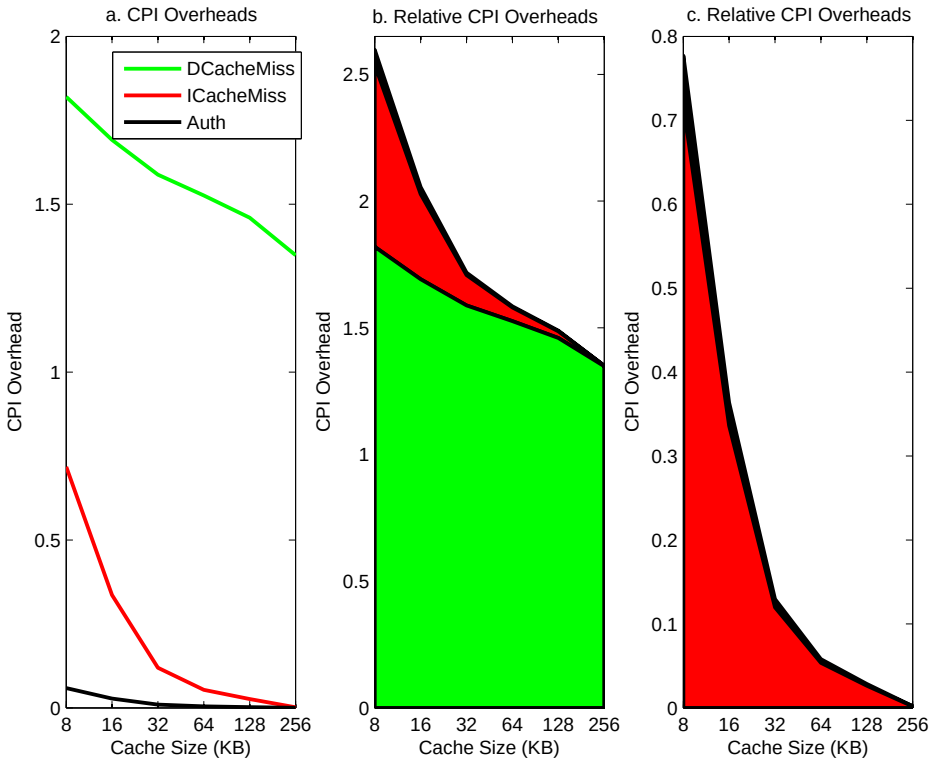


Figure 6: CPI Overheads due to instruction authentication, instruction and data cache misses for different cache sizes

the relative overhead due to instruction authentication to overhead due to instruction cache misses, which is taken as 1 for all block sizes. As observed from Figure 7 CPI overhead of instruction authentication is at most 30% of CPI overhead due to instruction cache misses. Considering CPI overheads due to other factors (such as data cache miss, data dependency, branch misspredicition), the overhead due to instructions authentication is still negligible for even the smallest cache blocks commonly used in embedded processors.

Some processors employ a technique in which the requested instruction starts executing without waiting for the other instructions in the same cache block to arrive from the memory. The caches that employ this technique are known as early-start caches. Furthermore, there are critical-word-first caches that fetch the requested instruction first from the memory. The proposed code authentication scheme, unfortunately, does not support these techniques since the processor has to wait for the entire instruction block to arrive in the cache and to authenticate before the execution of the requested instruction. We, therefore, explore the overhead incurred due to the fact the "early start" is not supported. In Figure 8, the area in blue shade represents the CPI overhead of lost clock cycles due to the "late start." The late start results in only a limited contribution to the overhead and this contribution increases with the block size.

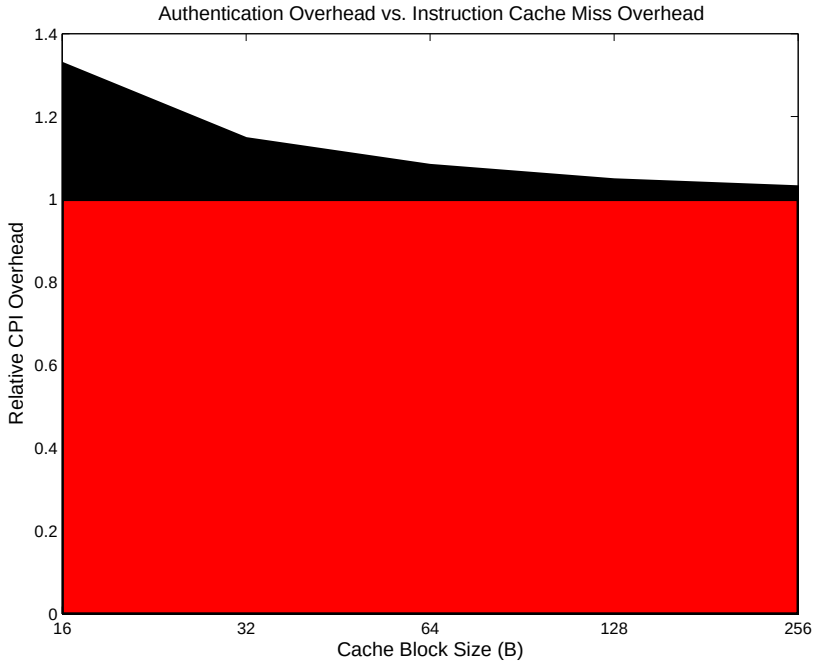


Figure 7: CPI Overheads due to instruction authentication normalized to CPI overhead of instruction cache misses for different block sizes

# 8 Simulation Results and Implementation Issues

## 8.1 Simulation Results

We used SimpleScalar [1] simulation tool to evaluate the overhead of the proposed scheme on the overall CPI (clock cycle per instruction). SimpleScalar is a cycle-accurate simulator for MIPS32 which is a RISC architecture used in both high-end and low-end processors. We used the out-of-order execution configuration for a conservative estimate for the overhead since out-of-order execution method usually achieves lower CPI values. In our experiments, five commonly employed benchmark programs are used: AES_RIJNDAEL, COMPRESS, DIJKSTRA, GO, and SHA-1[13].

In order to evaluate the effects of the proposed authentication scheme on a wide range of processors, we simulated the five benchmarks for several cache configurations where different cache and block sizes are used. We keep the configurations as modest as possible. For instance, we use only single-level cache organization where the largest cache size is 64 KB that is becoming common in embedded processors. Since the cryptographic computations are overlapped with memory access cycles, we only considered memory access latency for authentication tags as an overhead. We assumed a small, 16-entry authentication cache implemented as a FIFO buffer for holding recently accessed tags that takes approximately

[13]Benchmark programs are available at `http://www.seas.gwu.edu/~bhagiweb/cs211/SimpleScalar/SimpleScalarInstructions.html`

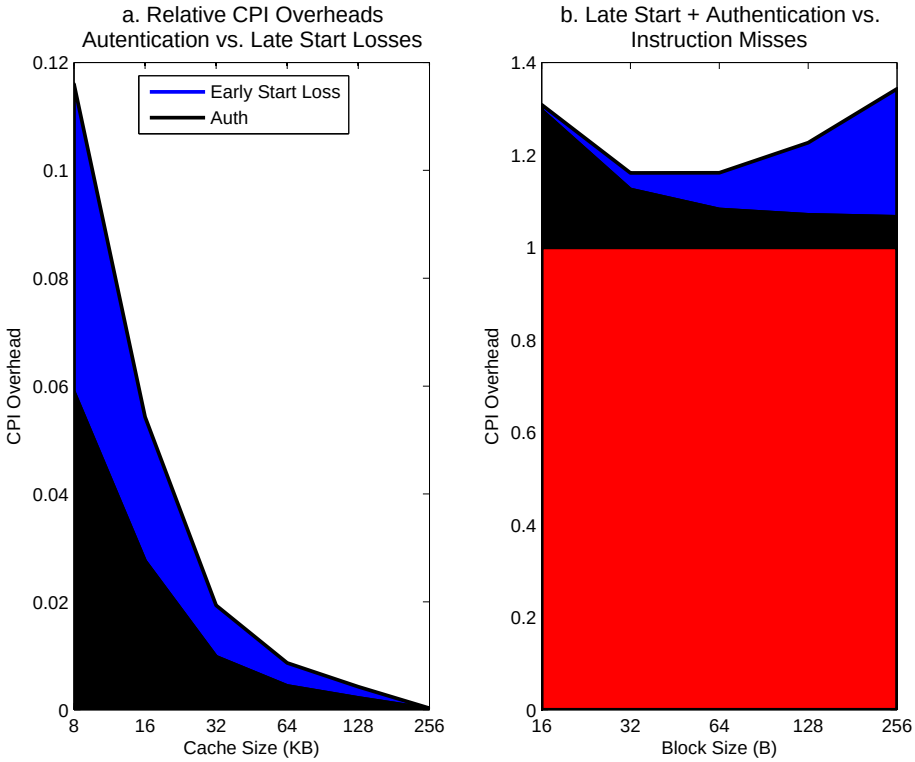


Figure 8: CPI Overheads due to instruction authentication (including losses due to late-start) normalized to CPI overhead of instruction cache misses for different block sizes

320 B on-chip memory space.

The simulation results are summarized in Table 3 where one easily observes that the overhead percentage of CPI even in modest cache configurations becomes negligible. For high-end performance processors, the overhead will virtually disappear.

## 8.2 Implementation Issues and Hardware Overhead

The most important design issue is whether the authentication mechanism can easily and inexpensively be integrated into a wide range of processor cores. Prohibitively high increase in hardware area is not acceptable while the latency of key generation mechanism (i.e. stream cipher *cf.* Figure 3) must be lower than the access latency of a memory block. As can be observed in Figure 3, the stream cipher is the most latency-sensitive and area demanding unit in the proposed authentication mechanism. The other units are merely simple modulo-2 adders, comparators, and relatively small amount of memory that we can use to store keys and tag values. In order to explore the feasibility of the proposed authentication and its overhead, we investigate hardware cost and latencies of state-of-the-art implementations of two ciphers; namely Trivium, a stream cipher proposed for eSTREAM (ECRYPT Stream

| Authentication Cache with 16 entry | | Cache Size (KB) | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 4 | 8 | 32 | 64 |
| Block (Size) (Bytes) | 16 | 67.92 | 53.77 | 49.01 | 28.03 | 13.74 |
| | 32 | 26.121 | 26.07 | 23.91 | 7.67 | 5.86 |
| | 64 | 11.87 | 13.49 | 12.87 | 3.81 | 2.57 |
| | 128 | 1.42 | 2.28 | 2.66 | 1.55 | 1.27 |
| | 256 | 0.17 | 0.39 | 0.35 | 0.55 | 0.56 |

Table 3: CPI overhead as percentage in simulation results

cipher project)[14], and AES that is originally a block cipher, but can be used to generate key bits. Four state-of-the art ASIC realizations of these ciphers are presented in Table 4, where the stream cipher implementation in [28] is both light-weight in terms of area and sufficiently fast to easily match the latency requirements of our application. Similarly, even the ASIC realizations of the stronger AES cipher[15] can be profitably used in a wide range of processor cores (from embedded to high end performance processors) for code authentication.

Table 4: State-of-the art designs suitable for stream cipher

| Design | Technology | Max. Frequency | Area | Throughput | bits/cycle |
|---|---|---|---|---|---|
| Trivium by Gaj et al. [28] | $90nm$ | 800 MHz | $\approx 5645$ | 51.2 Gpbs | 64 |
| AES by Satoh [37] | $0.11\mu m$ | 145 MHz | 12454 | 1.595 Gpbs | 11 |
| AES by Hodjat et al. [23] | $0.18\mu m$ | 606 MHz | 473000 | 77.6 Gpbs | 128 |
| AES by Northpole Eng. [17] | $0.25\mu m$ | 323 MHz | 26000 | 41.3 Gpbs | 127.86 |

The performance gap between the processor and memory technologies is one of the major issues in processor design since it tends to increase as the processor technologies improve much faster than memory technologies. Therefore, transferring a block of data or instruction between the memory and the cache is always one of the most expensive operations. The access latency of a memory block depends on the size of the memory block used for memory-to-cache transfers besides other factors. Depending on the bus width and the block size, bringing a block from memory to cache may typically take tens to hundreds of clock cycles. The access latency of memory blocks in terms of the number of clock cycles used in the SimpleScalar [1] simulator is given in the first row of Table 5 for different block sizes. The proposed authentication scheme requires that the same number of bits as the block size be generated by the ciphers. Table 5 shows that all designs except for the one in [37] generate the number of key bits sufficient for both instruction authentication and encryption. The design in [37] can be profitably used for embedded systems where typical memory block sizes do not exceed 64 B. The design in [28], which is very fast and extremely area efficient, easily matches the requirements of both embedded and of high-end performance processors.

[14]see `http://www.ecrypt.eu.org/stream/` and `http://www.ecrypt.eu.org/stream/triviumpf.html`

[15]AES offers 128-bit security versus Trivium's 80-bit.

Table 5: Number of clock cycles required by each design to generate key bits when operated at frequencies up to maximum

| Design | 1 bit | 1 B | 16 B | 32 B | 64 B | 128 B | 256 B |
|---|---|---|---|---|---|---|---|
| Memory Latency in SimpleScalar | - | - | 26 | 34 | 50 | 82 | 146 |
| [28] | 21 | 21.125 | 23 | 25 | 29 | 37 | 53 |
| [37] | 0.086 | 0.678 | 11 | 22 | 44 | 88 | 176 |
| [23] | 0.00781 | 0.0625 | 1 | 2 | 4 | 8 | 16 |
| [17] | 0.00782 | 0.0626 | 1 | 2 | 4 | 8 | 16 |

The high-end performance processors work at higher frequencies and therefore may put higher demands on our key generation mechanism. Working at their highest possible clock frequencies, three designs in [28, 23, 17] can still provide a sufficient number of key bits for a processor working at frequencies as high as 2 GHz (*cf.* Table 6). Note that for processors working at very high frequencies, memory latencies can be much higher than our modest estimate derived from the SimpleScalar simulator. Even though the design in [23] is excessive in chip area usage, its overhead is negligible for high-end processors whose transistor counts are expressed in hundreds of millions coming close to a billion.

Table 6: Number of clock cycles required by each design to generate key bits at clock frequencies 1 GHz/2 GHz

| Design | 16 B | 32 B | 64 B | 128 B | 256 B |
|---|---|---|---|---|---|
| [28] | 28.75/57.5 | 31.25/62.5 | 36.25/72.5 | 46.25/92.5 | 66.25/132.5 |
| [23] | 1.65/3.3 | 3.3/6.6 | 6.6/13.2 | 13.2/26.4 | 26.4/52.8 |
| [17] | 3.09/6.2 | 6.2/12.4 | 12.4/24.8 | 24.8/49.6 | 49.6/99.2 |

In summary, a quick overview of current technology and state-of-the-art for cryptographic algorithm realizations reveals that the proposed code authentication scheme can easily be integrated into a wide range of processor cores with a negligibly low overhead in area and time.

Another practical issue is faults occurring in memory, registers, data bus, and control circuitry that corrupt the bits in instructions, data and of special interest to us in authentication tags. The faults introduced to computation by mother nature or adversaries are shown to result in many undesired and dangerous situations where secrets are compromised [6, 24, 10] and the computation is corrupted or stalled. Therefore, there is a plethora of published works (e.g. [25, 40, 16, 35, 20]) that propose solutions to protect different aspects of the computation. Only foreseeable result due to faults in authentication tags is that the computation is interrupted as the hardware refuses to execute the unauthenticated instructions. This can be considered as a security flaw since adversary may mount a denial of service attack by introducing faults in the authentication tags. Therefore, authentication tags should

be protected against faults, adversarial or otherwise. However, the faults in authentication tags do not require a special treatment since corrupted bits in data and instructions normally results in similar consequences. For instance, a corrupted bit in an instruction can result in an undefined instruction or access to restricted parts of the memory; each case an exception is thrown and execution most probably stops. The advantage of the proposed scheme is that any mechanism deployed for the protection of data and instructions naturally protects the authentication tags since they are part of the program state and stored and treated in the same manner. Another advantage of using authentication tags is that any adversarial fault attack is immediately detected due to authentication circuitry. The protection against adversarial faults and fault attacks must be approached from a holistic perspective so that every aspect of the computation is considered. This is an active research area in its infancy and naturally deserves serious treatment that is beyond the scope of this work.

# 9 Previous Work and Comparison

There is plethora of work on secure and trusted execution of programs where the primary goal is to provide the programs with a secure environment free from the interference of other malicious programs. The common perception is that software-only solutions are inadequate and that hence hardware support is necessary. Furthermore, the operating system that implements the core protection mechanism is overly complex and it is not possible write a bug-free OS that is safeguarded against attacks. Major microprocessor manufacturers, (Intel, AMD, ARM) already introduced hardware extensions to their processor cores that allow isolated execution of programs [13, 15, 3]. This can be achieved by making the portions of memory, of cache, of TLB used by a program inaccessible to other programs. The techniques proposed in [32, 33] deals with performance problems in isolated execution of security-sensitive codes by minimizing trusted code base and by proposing some hardware extensions. They also allow fast and fine-grained attestation of the code executed. However, both the code base that manages the isolated execution mechanism and the code running in isolated environment have certain privileges (e.g. accessing sensitive information) and therefore must be trusted. The proposed code authentication scheme provides this trust by authentication every code block before their execution in an efficient manner.

Another line of work is concerned with encryption and authentication of frequently and dynamically changing data used by programs in execution. Theorems 1 and 2 imply that no adversary can *generate* an alternative value for a block. He can, however, substitute a data block with an old (and already authenticated) data block. This is not a breach of security for authenticated code, since the code is assumed to be static (and each block of code is authenticated *together* with its address), but for dynamic data this is a security breach. Authenticating data is not in the scope of this work, but since data can have an impact on the execution of a program, we describe how our code authentication scheme can work together with previously proposed memory authentication schemes — in particular the schemes presented in the papers [11, 19, 12, 42].

In [19] Gassend et al. propose a memory authentication scheme based on Merkle trees [34]. To authenticate an arbitrarily large untrusted RAM memory, an $m$-array tree structure is built up. Each node of the tree contains collision resistant hashes of each of the $m$ children.

The leaves of the tree contain the actual data. Parent nodes contain hash values of data *or other hash values*. The root of the tree is kept on-chip in a trusted register of constant size, all other nodes are kept in main memory or cache. To authenticate a block of data (or an inner node) all hash values from the data to the root of the hash-tree are checked. To update data, each hash value for the data block to the root has to be recalculated. This scheme is secure, since all modifications to the memory require a modification of the root of the tree, which is kept in a trusted register, thus no unauthenticated modifications can take place. The worst case time complexity of each read or write operation is $O(\log_m(N))$ (for balanced hash-trees), where $N$ is the size of the memory. However, the scheme is improved by observing that nodes that are already in trusted L1 cache do not need re-authentication. Re-authentication of dirty cache-lines is only done when the cache-lines are flushed to memory. The memory overhead of the scheme is $1/(m-1)$.

In [11] Clarke et al. propose a scheme which keeps (the hash values of) logs of all read and write operations. The memory can then be authenticated off-line by comparing the logs with the actual content of the memory. This scheme is inspired by incremental hashing by Bellare et al. [4]. Clarke et al. introduce the new concept of *multiset hash functions*, which are families of hash functions that map a multiset (a set with possible repetitions) to a constant size hash value. They give three constructions of multiset hash functions which are incremental — that is: given two multisets and their hash values, it is efficient to compute the hash value of the multiset-union of the two multisets. The idea for memory authentication is to keep logs of all read and write operations. Log entries contain address, data, and timestamp triples. Since the logs can become arbitrarily large, only two multiset hashes are kept: one for read and one for write operations. When authentication is requested *all* memory locations have to be compared with the logs. This gives a very inefficient authentication of $O(N)$, but read and write operations require only a small constant overhead (adding entries to the read and write logs). The only memory requirement for this scheme are the two multiset hash values of the logs. Since authentication is very expensive, it is only done when data is exported out of a program execution environment (e.g. microprocessor).

In [12] Clarke et al. propose a hybrid protocol with the aim of balancing the drawbacks and benefits of the two protocols described in [11] and [19]. The overhead of this scheme tends to a constant as the number of instructions between critical instruction grows.

The state-of-the-art in memory encryption/authentication implementations is introduced by Yan et al. in [42], where Galois/Counter Mode of operation (GCM) is used to reduce authentication latency and overlap it with memory accesses (similar to our scheme in this respect). Using GCM encryption and authentication together resembles our authentication scheme; however our construction is more general and more flexible in the sense that it allows different implementation alternatives. They provide implementation results of their schemes where only 4% IPC degradation is reported which is a significant improvement over classical hash- or MAC-based schemes. As the work targets high-end performance processors, the implementation results are given for a system with a rich configuration; e.g. 1 MB L2 cache with 64-byte blocks, 128-bit system bus that connects memory and processor etc. The size of the cache and cache blocks have a decisive effect on the miss rates and a cache of 1 MB as used in the experiments of [42] results in a very low miss rate and hence the cryptographic operation for tag computation becomes extremely infrequent. A system bus of 128-bit perfectly matches 128-bit Galois field multiplication used in the GCM authentication.

Another issue is that the length of the authentication tag is a mere 64-bit which does not provide a sufficient security since finding collusions is not difficult for 64-bit authentication tags[16]. Although 64-bit authentication tags can be used for frequently changing data blocks, its use for static instruction blocks is definitely not advisable. The scheme uses 32 KB on-chip cache for counter values, which may not be accommodated in embedded systems. Furthermore, the authentication scheme must be implemented with the encryption scheme, which is an additional burden for cases where the encryption is not needed. And finally, the scheme relies on block ciphers which are considered to be slower than stream ciphers that can be profitably used for their speed and low resource requirements.

It is natural to ask how the schemes proposed in [11], [19], and [12] relate to our scheme. These schemes solve the problem of "checking if the untrusted RAM behaves like valid RAM [12]." The aim of these papers is to ensure that only the main processor can modify the memory. However, the processor will authenticate any modification that is made to the memory by code which is running on the processor. There is no mechanism to prevent a virus, for instance, to modify data (or code) in the memory. To avoid this problem two things are needed: 1) code must be authenticated before it is executed (in particular when it is loaded into the memory), and 2) modification of code must be prevented. The schemes in [11, 19, 12, 42] can be used in conjunction with a TPM to authenticate code at the time it is loaded, by, e.g. letting the TPM verify a digital signature of the code before the code is executed. This, however, will increase the load time of a program considerably. To prevent modification of code, write protection mechanisms should be added to the schemes in [11, 19, 12, 42]. However, such a scheme will always inherit the non-constant authentication overhead from these techniques.

In the schemes of [11] and [19] the validation of any block of memory involves the trusted on-chip hash value(s). In particular, after loading a new program into memory the scheme in [19] has to compute the entire sub-hash-tree covering the code of the program in order to validate the first instruction. For large programs this will give relatively long loading times. In [11] all verifications involves verifying the entire memory.

The scheme in [11] is specially tuned for dynamic data, since it keeps a log of read and write operations. However, verification of any block of memory in [11] involves hashing of the entire memory. While keeping read and write logs is a good solution for dynamic data, it clearly introduces unnecessary overhead if the data is static.

Since our scheme does not face the difficulties of dynamically changing memory, we can avoid the associated overhead. Each authentication operation in our scheme only introduces a small constant overhead which can mostly be hidden in the memory access operation. Furthermore, as discussed in Section 3, our scheme can be used directly to verify that the code which is loaded and executed is authentic, as well as to protect the intellectual property of software manufacturers.

The above discussion shows how our scheme and the schemes in [11, 19, 12, 42] complement each other. Our scheme can authenticate a program and the static data that is loaded into the memory, while the schemes in [11, 19, 12, 42] can guarantee that dynamic data is created by the application and not the adversary. While our scheme can be used for code

[16]The GCM schemes allows up to 128-bit MACs. However, a longer tag than 64-bit will negatively affect the IPC degradation in [42].

and static data authentication for efficiency reasons, the previous schemes in [11, 19, 12, 42] can be used for dynamic data authentication once the program execution starts.

Lee et al. in [30] propose a technique for code authentication based on AES-MAC computation along with encryption. They report that the latency due to authentication is 100 clock cycles and that overall performance degradation is around 1%. One major issue with this technique is that the high latency of authentication tag computation is not overlapped with memory access operation. Another, probably more important issue, is that performance figures are obtained for a very large L2 cache of 2MB for which instruction misses are extremely infrequent and therefore instruction authentication occurs extremely infrequently as well. However, for smaller caches used in embedded processors, the performance penalty can be prohibitively high. The last issue with the technique in [30] is that 16 bytes of an instruction block is used to store the authentication tag for the rest of the instruction block. Since the effective size of the instruction block is actually reduced, the miss rate will increase. Furthermore, the technique may not be applicable for cache lines of smaller sizes; namely 16 B and 32 B common in embedded processors. In contrast to the technique in [30], our method does not affect the instruction miss rate, it can be applied in embedded systems with much smaller cache and block sizes, most of the cryptographic computations can be overlapped with memory access cycles, and our authentication cache can take advantage of spatial locality.

Yang et al. in [43] propose an efficient memory encryption technique where cryptographic computation is overlapped with memory access cycles. This technique deals with the encryption/decryption of dynamically changing data and data/instruction authentication is not addressed. They report 1.28% performance penalty for a system with 256 KB L2 unified cache of 128 B cache lines where an additional 64 KB on-chip storage is required.

Any comparison of our scheme with the previous work is not fair due to the following reasons: i) different problems are addressed (trusted code execution through instruction authentication vs. memory encryption and authentication), ii) different classes of processors are targeted (a wide range of processors versus high-end performance processors), and iii) different simulation environments (simulation tool and benchmarks) are used. Our two basic claims are that the code authentication is a different problem than memory encryption and authentication and that we do not have to suffer high hardware and timing complexities of the latter since the former can be solved in a much more efficient way. Having said that the comparison is not fair, it may still be useful to display different aspects of our scheme vis-à-vis with those of the previous schemes as we do in Table 7.

As observed in Table 7, the proposed scheme can provide code and static data authentication of sufficient security level with a very low overhead both in time complexity and area usage. Its low cost nature renders itself to be used even in low-cost resource constrained embedded processors for which the estimated cost of the previous schemes are not affordable. Conversely, the latency overhead of the proposed scheme will virtually disappear for high-end performance computers.

Table 7: An unfair comparison with the previous schemes

| Aspects | [30] | [43] | [42] | **proposed** |
|---|---|---|---|---|
| Cache size | 2 MB | 256 KB | 1 MB | 64 KB (max.) |
| Block size | 64 B | 128 B | 64 B | 16 B and up |
| Cache levels | 2 | 2 | 2 | 1 |
| On-chip memory for tags | NA | 64 KB | 32 KB | 320 B (max.) |
| Reported overhead (best) | 1% | 1.28% | 4% | 0.56% |
| Dynamic data authentication | No | No | Yes | No |
| Separate Authentication and Encryption | Yes | NA | No | Yes |
| Cryptographic Primitive | Block Cipher | Stream Cipher | Block Cipher | Stream Cipher |
| Authentication Tag Size | 128 | NA | 64 | 128 |

# 10 Conclusion

In this paper we presented a code authentication scheme with low overhead that is tightly integrated into a processor architecture to facilitate on-the-fly code and static data authentication. The presented scheme builds on the previously proposed idea of hiding the computational latency of encryption behind memory access latencies. The security of the scheme is established using message authentication codes, based on efficient universal families of hash functions, which provide security when used with one-time pad encryption. Furthermore, since the proposed authentication technique manages to hide the latency, it is suited to be used to ensure the integrity of code and static data blocks encrypted with efficient stream ciphers which otherwise are open to many forms of attacks. Hence, the proposed authentication technique enables off-the-critical-path code encryption. The performance analysis shows that the presented architecture bears little overhead for even modest cache sizes. Our simulation results confirms our claims on the reduced overhead in time and space complexities.

# References


[1] The simplescalar tool set. Available at `http://wwww.simplescalar.com/`.

[2] Kazumaro Aoki and Helger Lipmaa. Fast implementations of AES candidates. In *AES Candidate Conference*, pages 106–120, New York City, USA, 13–14 April 2000.

[3] ARM. *TrustZone Technology Overview.* `http://www.arm.com/products/security/trustzone/`.

[4] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Desmedt [14], pages 216–233.

[5] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1999.

[6] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *EUROCRYPT*, pages 37–51, 1997.

[7] Gilles Brassard. On computationally secure authentication tags requiring short secret shared keys. In *CRYPTO'82*, Lecture Notes in Computer Science, pages 79–86. Springer-Verlag, 1982.

[8] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[9] Benoît Chevallier-Mames, David Naccache, Pascal Paillier, and David Pointcheval. How to disembed a program? In Marc Joye and Jean-Jacques Quisquater, editors, *CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 441–454. Springer-Verlag, 2004.

[10] H. Choukri and M. Tunstall. Round reduction using faults. In L. Breveglieri and I. Koren, editors, *2nd International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'05)*, pages 13–24, 2005.

[11] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, 2003.

[12] Dwaine E. Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE Symposium on Security and Privacy*, pages 139–153. IEEE Computer Society, 2005.

[13] Intel Corporation. *LeGrande technology preliminary architecture specification.* Intel Publication no. D52212, May 2006.

[14] Yvo Desmedt, editor. *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[15] Advanced Micro Devices. *AMD64 virtualization: Secure virtual machine architecture manual.* AMD Publication no. 33047 rev. 3.01, May 2005.

[16] D. Pradhan ed. *Fault Tolerant Computing – Theory and Techniques*, volume 1. New Jersey: Prentice-Hall, 1st edition, 1986.

[17] North Pole Engineering. AES core. `http://www.hardware-ciphers.com/en/aes/asic-unrolled.html`.

[18] Kris Gaj and Pawel Chodowiec. Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays. In David Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2001.

[19] Blaise Gassend, G. Edward Suh, Dwaine E. Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity. In *Proceedings of Ninth International Symposium of High Performance Computer Architecture (HPCA 2003)*, pages 295–306, February 2003.

[20] Gunnar Gaubatz and Berk Sunar. Robust finite field arithmetic for fault-tolerant public-key cryptography. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *FDTC*, volume 4236 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2006.

[21] Oded Goldreich. Secure multi-party computation. Working Draft, Verison 1.1, 1998. `citeseer.ist.psu.edu/article/goldreich98secure.html`.

[22] J. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc. (Elsevier), 3rd edition, 2002.

[23] Alireza Hodjat and Ingrid Verbauwhede. Speed-area trade-off for 10 to 100 Gbits/s throughput AES processor. In *2003 IEEE Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[24] Marc Joye, Arjen K. Lenstra, and Jean-Jacques Quisquater. Chinese remaindering based cryptosystems in the presence of faults. *J. Cryptology*, 12(4):241–245, 1999.

[25] Albert L. Hopkins Jr. and T. Basil Smith III. The architectural elements of a symmetric fault-tolerant multiprocessor. *IEEE Trans. Computers*, 24(5):498–505, 1975.

[26] Jens-Peter Kaps, Kaan Yüksel, and Berk Sunar. Energy scalable universal hashing. *IEEE Trans. Computers*, 54(12):1484–1495, 2005.

[27] K. Kaukonen and R. Thayer. A stream cipher encryption algorithm "ARCFOUR", internet engineering task force (IETF) internet draft, July 14 1999. `http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt`.

[28] K.Gaj, G. Southern, and R. Bachimanchi. Comparison of hardware performance of selected phase ii eSTREAM candidates. State of the Art of Stream Ciphers Workshop (SASC 2007), February 1 2007. `http://www.ecrypt.eu.org/stream/papersdir/2007/026.pdf`.

[29] Hugo Krawczyk. Lfsr-based hashing and authentication. In Desmedt [14], pages 129–139.

[30] Ruby B. Lee, Peter C. S. Kwan, John Patrick McGregor, Jeffrey S. Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA*, pages 2–13. IEEE Computer Society, 2005.

[31] Daihyun Lim, Jae W. Lee, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Extracting secret keys from integrated circuits. *IEEE Trans. VLSI Syst.*, 13(10):1200–1205, 2005.

[32] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In Joseph S. Sventek and Steven Hand, editors, *EuroSys*, pages 315–328. ACM, 2008.

[33] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go?: recommendations for hardware-supported minimal tcb code execution. In Susan J. Eggers and James R. Larus, editors, *ASPLOS*, pages 14–25. ACM, 2008.

[34] Ralph C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Electrical Engineering, Stanford, 1979.

[35] Arash Reyhani-Masoleh and M. Anwar Hasan. Towards fault-tolerant cryptographic computations over finite fields. *ACM Trans. Embedded Comput. Syst.*, 3(3):593–613, 2004.

[36] Marcin Rogawski. Hardware evaluation of estream candidates: Grain, lex, mickey128, salsa20 and trivium. State of the Art of Stream Ciphers Workshop (SASC 2007), February 1 2007. `http://www.ecrypt.eu.org/stream/papersdir/2007/025.pdf`.

[37] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact rijndael hardware architecture with s-box optimization. In Colin Boyd, editor, *ASIACRYPT*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[38] D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC (Taylor Francis Group), 3rd edition, 2006.

[39] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In Utpal Banerjee, Kyle Gallivan, and Antonio González, editors, *ICS 2003*, pages 160–171. ACM, 2003.

[40] Berk Sunar, Gunnar Gaubatz, and Erkay Savas. Sequential circuit design for embedded cryptographic applications resilient to adversarial faults. *IEEE Trans. Computers*, 57(1):126–138, 2008.

[41] Trusted Computing Group, Incorporated. *TCG Software Stack (TSS), Specification Version 1.2, Level 1. Part1: Commands and Structures*, January 6 2006. `https://www.trustedcomputinggroup.org/specs/TSS/TSS_Version_1.2_Level_1_FINAL.pdf`.

[42] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. In *ISCA*, pages 179–190. IEEE Computer Society, 2006.

[43] Jun Yang, Lan Gao, and Youtao Zhang. Improving memory encryption performance in secure processors. *IEEE Trans. Computers*, 54(5):630–640, 2005.

[44] Youtao Zhang, Jun Yang, Yongjing Lin, and Lan Gao. Architectural support for protecting user privacy on trusted processors. *SIGARCH Computer Architecture News*, 33(1):118–123, 2005.

**Proof of Theorem 2**

**Proof 1** *Let $M$, $M'$ be distinct members of the domain $A$ with equal lengths. We are required to show that*

$$Pr\left[\mathsf{PR}_K(M) \oplus \mathsf{PR}_K(M') = c\right] = 2^{-w} \ .$$

*Note that since we are working over a finite field $GF(2^w)$ the exclusive-or operation is identical to an addition operation and we may use the symbols '$\oplus$' and '$+$' interchangibly. Expanding the terms inside the probability expression, we obtain*

$$\sum_{i=1}^{n/2}(m_{2i-1}+k_{2i-1})(m_{2i}+k_{2i}) + \sum_{i=1}^{n/2}(m'_{2i-1}+k_{2i-1})(m'_{2i}+k_{2i}) = c \pmod p \ .$$

*The probability is taken over uniform choices of $(k_1, k_2, \ldots, k_n)$ with each $k_i \in GF(2^w)$. Since $M$ and $M'$ are distinct, $m_i \neq m'_i$ for some $1 \leq i \leq n$. Addition and multiplication in $GF(2^w)$ are commutative, hence there is no loss of generality in assuming $m_2 \neq m'_2$. Hence we need to prove that for any choice of $k_2, k_3, \ldots, k_n$ that*

$$Pr_{k_1 \in GF(2^w)}\left[(m_1+k_1)(m_2+k_2) + \sum_{i=2}^{n/2}(m_{2i-1}+k_{2i-1})(m_{2i}+k_{2i}) + \right.$$
$$\left.(m'_1+k_1)(m'_2+k_2) + \sum_{i=2}^{n/2}(m'_{2i-1}+k_{2i-1})(m'_{2i}+k_{2i}) = c \pmod p\right] \leq 2^{-w}$$

*Let*

$$y = \sum_{i=2}^{n/2}(m'_{2i-1}+k_{2i-1})(m'_{2i}+k_{2i}) + \sum_{i=2}^{n/2}(m_{2i-1}+k_{2i-1})(m_{2i}+k_{2i}) \ .$$

*Rewriting the identity inside the probability yields*

$$k_1(m_2+m'_2) = y + c + m_1(m_2+k_2) + m'_1(m'_2+k_2) \pmod p \ .$$

*Since $m_2 \neq m'_2$, the term $(m_2+m'_2)$ cannot be zero and its inverse in $GF(2^w)$ exists. Hence there is exactly one $k_1 \in GF(2^w)$ satisfying the equation, which is*

$$k_1 = (m_2+m'_2)^{-1}\left(y + c + m_1(m_2+k_2) + m'_1(m'_2+k_2)\right) \pmod p \ .$$

*Therefore,*

$$Pr\left[\mathsf{PR}_K(M) \oplus \mathsf{PR}_K(M') = c\right] = 2^{-w} \ .$$