Lightweight Software Encryption for Embedded Processors

Thomas Hiscock, Olivier Savry
CEA, LETI, MINATEC Campus,
F-38054 Grenoble, France.
Email: thomas.hiscock@cea.fr, olivier.savry@cea.fr

Louis Goubin
Laboratoire de Mathématiques de Versailles,
UVSQ, CNRS, Université Paris-Saclay,
F-78035 Versailles, France.
Email: louis.goubin@uvsq.fr

***Abstract*—Over the last 30 years, a number of secure processor architectures have been proposed to protect software integrity and confidentiality during its distribution and execution. In such architectures, encryption (together with integrity checking) is used extensively, on any data leaving a defined secure boundary.**

In this paper, we show how encryption can be achieved at the instruction level using a stream cipher. Thus encryption is more lightweight and efficient, and is maintained deeper in the memory hierarchy than the natural off-chip boundary considered in most research works. It requires the control flow graph to be used and modified as part of the off-line encryption process, but thanks to the LLVM framework, it can be integrated easily in a compiler pipeline, and be completely transparent to the programmer.

We also describe hardware modifications needed to support this encryption method, the latter were added to a 32 bit MIPS soft core. The synthesis performed on a Altera Cyclone V FPGA shows that encryption requires 26% of extra logic, while slowing-down execution time by an average of 48% in the best setting.

I. Introduction

An increasing number of applications require processor architectures that are both lightweight and able to preserve software confidentiality. The historical motivation was a purely economic one: protect intellectual property and prevent illegitimate duplication. Nowadays, software confidentiality is rather seen from a security perspective, to prevent reverse-engineering. The program being more resilient to analysis, the effort needed to discover weaknesses is increased, and critical patches can be deployed without fear of zero day exploits.

If one can modify the hardware, software encryption is a well established way to achieve this confidentiality. The program is encrypted using a regular cryptographic primitive, and decryption is done using a hardware implementation of the decryption algorithm in an assumed secure area. The latter is close to the processor executing the software, at least on the same chip. Aside from the confidentiality property, encryption alone brings other interesting properties from a security perspective:

- as each target have different encryption keys, shell code design is harder, thus exploits cannot be deployed quickly on a large scale.
- It can be used as a building block to provide control flow integrity, as shown recently with the SOFIA [1] architecture.

Current secure processor architectures are mostly concerned with protecting program and data stored on off-chip memories like Flash, Dynamic RAMs (DRAMs), the chip area being assumed safe. For this reason, and also for performance concerns, decryption is usually done at a cache (level 1, or level 2) memory boundary. As a consequence, data are decrypted by chunks made of one or more cache lines (32B, 64B, 128B or even more). This way, the latency of the decryption algorithm can be almost completely hidden, by overlapping the decryption with memory fetches on a cache miss.

However, to the best of our knowledge, very few works (but [2]) provide methods to achieve a finer encryption granularity, namely at the instruction level. Yet, it is required for applications in which the target processor either do not have cache, or needs to maintain encryption deeper in the memory hierarchy.

In this work we show how encryption can be done at the instruction level using stream ciphers, which are known to be very lightweight and efficient. It requires the control flow graph of the program to be used and restructured as part of the encryption process. The proposed method, developed as a LLVM [3] backend pass, can encrypt almost any given machine code and do not require any modification from the programmer but adding a compiler flag. We stress that we only describe a method for encrypting the software, not verifying its integrity. Of course, integrity is as much as important as encryption in secure processors [4], but its treatment can be somewhat orthogonal, so we decided to focus on the encryption mechanism.

The hardware support, including decryption hardware (based on the Trivium [5] stream cipher) is added into MIPS soft core and deployed on a low cost Altera Cyclone V Field Programmable Gate Array (FPGA). On this small core, the encryption mechanism requires only 26% of extra logic. The execution slowdown is highly dependent on the compilation profile. In the best setting (performance optimized programs), we measured an average slowdown of 48%, across all benchmarks slowdown is between 29% up to 193%. These results illustrate that a very lightweight and efficient encryption can be achieved to target real-world applications on constrained processors.

The rest of this paper is structured as follow: Section II presents in more details the security model used in this

work, followed by a survey of related work in Section III. Our software encryption process as well as its integration in the LLVM framework is described in Section IV. Then, the processor modifications needed to support encryption, and implementation of common software abstractions like exceptions, context switches are presented in Section V. We conclude this paper by an evaluation of our method in Section VI. The security of this solution is discussed, and result from our practical implementation on FPGA are analyzed.

II. Security Model

In this work we consider a standard System on Chip (SoC) system, with a single processor, as the one shown Figure 1. The processor itself may or may not have instruction and data caches. Secure processors usually draw an insecure boundary at the off-chip memories interfaces. Beyond this boundary, it is assumed that any data can be observed, or tampered with.

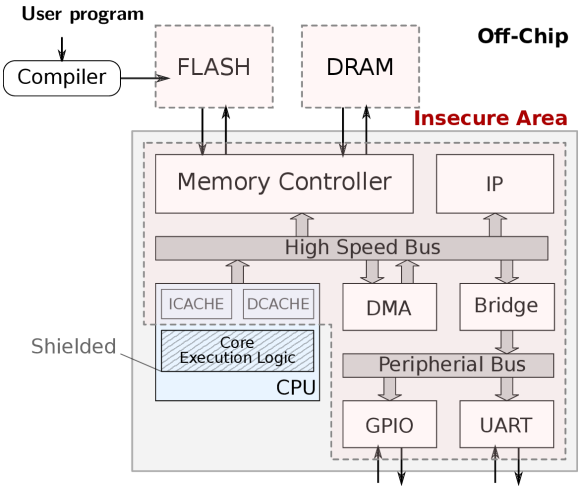


Fig. 1. A typical system considered in this work, with the on-chip insecure area drawn

Indeed, many popular attacks showed that memory can be easily extracted on a wide range of devices, even with cheap hardware. As an example, "Cold boot" family attacks exploit data persistence in DRAMs [6]: after system reboot some critical part of memory can then be recovered. Direct Memory Access (DMA) components were successfully used to obtain read or write access to CPU's memory through some user accessible peripherals (e.g., Firewire [7]). Even direct probing using FPGA or low cost modchips [8] is feasible on external buses like PCI express.

On the other hand, attacking the internals of a processor, say, reading a register value at a given time, is far more challenging and requires advanced physical attacks techniques as well as expensive equipments [9], [10].

In this work, our goal is to protect the confidentiality of a given machine code. Our insecure boundary is moved deeper inside the chip, between the processor memory interfaces and its execution logic (caches are also considered as insecure). Formally, an adversary is allowed to:

- read any data stored into off-chip memories (the latter will be ciphered),
- read instructions located into the instruction cache or any on-chip memory.

For this purpose, we assume that the CPU's execution logic (Figure 1) is shielded, physical attacks cannot be performed, such that cryptographic operations can be done safely inside the core. This shielded region includes CPU's internal state, like the program counter (PC), registers, etc...

This work is primarily concerned with software protection. The programs executed on the device are assumed «safe», in the sense that they do not store critical data in memory, or if so, manipulate them using a dedicated secure coprocessor.

III. Related work

Obfuscation techniques are a well-known class of software-only counter measures [11], but cannot achieve provable security even for restrained models [12] without some secure hardware. Heuristic techniques have proven to increase the time and effort needed to reverse a program, but can be defeated by an experimented adversary. Furthermore, the overhead on both program size and execution time is quite high: depending on obfuscation level, factors between $\times 10$ and $\times 100$ are common [13].

The use of software encryption in processors dates back to Best [14], [15], who proposed a series of patents that made up the basis of the Dallas DS5002 [16] secure processor. Early versions of the Dallas DS5002 were defeated by a famous attack performed by Kuhn [4]. He managed to inject instructions and monitor I/O to build a malicious code capable of dumping the whole memory.

Since, number of researchers proposed hardware-assisted memory encryption [17]–[21]. A block cipher is used as encryption primitive to perform data authentication and decryption when accessing data from insecure external memory. The plain content is then placed in a processor-close memory, assumed free from tampering (local RAM or a cache).

Perhaps the closest approach to ours is Instruction Set Randomization (ISR) [2], [22], [23], though mainly designed to prevent code injection. It was shown in [2] that ISR can also be used to prevent reverse engineering. They added an additional processor instruction called `rev` to randomize the instruction set on demand. They implemented this software encryption using the Trivium stream cipher on top of a Leon2 (SPARC V8) core. Compared to this work and more generally ISR techniques, our solution do not need any instruction set extension, so it is more transparent to the programmer.

IV. Static Code Encryption with Stream Ciphers

A. Background on Stream Ciphers

Stream ciphers are an efficient class of pseudorandom generators. Unlike block ciphers which provide a fixed-length permutation, they can produce an arbitrary long pseudorandom sequence. Formally speaking, a stream cipher is specified by two functions:

- `init` : $\mathcal{K} \times \mathcal{IV} \rightarrow \mathcal{S}$, which generates an initial state from a secret key and an Initialization Vector (IV). The IV can be made public, while the secret key must be kept private.
- `genBits` : $\mathcal{S} \rightarrow \mathcal{S} \times \mathcal{C}$, which produces the next state and a pseudorandom output.

Once initialized with `init` and an arbitrary IV, pseudorandom bits can be generated on demand and used as a one-time pad to provide an encryption scheme, as shown in Figure 2. The IV has to be transmitted with the ciphertext to allow the receiver to decrypt. We stress that IVs have to be uniformly distributed to guarantee the full security of the scheme under chosen plaintext attacks (IND-CPA). Furthermore non uniform IV (using counter mode-like construction) might lead to reduced attack complexity through time space trade-off attacks [24].

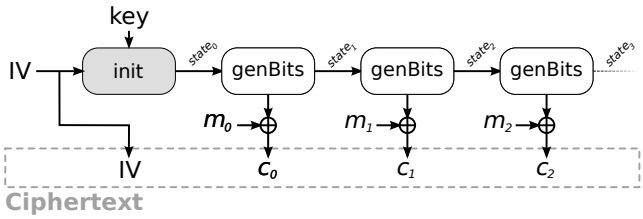


Fig. 2. Encryption using a stream cipher

For most stream ciphers, the `init` function is a costly operation while the `genBits` function is quite fast. As a consequence, stream ciphers are good for generating very long pseudorandom sequences.

Let us note that a stream cipher can be constructed from a block cipher using the output feedback mode of encryption [25]. However, dedicated stream ciphers are far more efficient. In 2008, the eStream [5] competition selected a set of recommended stream ciphers. For the hardware profile, three were selected: Trivium, Grain and Mickey.

B. Why Stream Ciphers?

Block ciphers are a good established way achieve encryption in secure processor architectures [26]. The counter mode of encryption is a good fit for encrypting a processor address space. Indeed, the base address of the data (or a block of data) can be used as counter value, and, unless virtual memory is used [21], [27], counter value uniqueness is guaranteed.

However, block ciphers suffer from two limitations for being used to encrypt at the instruction level. The first one is that block ciphers are intrinsically fixed-length permutations, and for security reason, the minimum recommended block size is 128 bits. On the other hand, in many instruction set architectures, instructions almost never take more than 32 bits [28]. To exploit the full throughput of the block cipher, some sort of complex instruction padding has to be designed.

The second limitation is that the decryption primitive has to be able to work at CPU's execution speed. Of course, a fully pipelined implementation of a block cipher would meet this requirement, but may also significantly increase the hardware footprint. As an illustration, the fully pipelined AES implementation from OpenCores[1], is 3 times the size of our base MIPS processor.

On the other hand, instruction execution is most of the time a very sequential process, at least on in-order processor architectures. As already mentioned, stream ciphers, once initialized, are quite efficient to generate random bits. For this reason they seem to fit quite well with the job of encrypting software at instruction granularity. Once initialized, a stream cipher is usually able to decrypt one instruction per cycle, whereas a block cipher would require several rounds between each instruction.

C. Encrypting Machine Code

Because of the stateful nature of stream ciphers, it is not straightforward to use them for encrypting instructions. Let first remark that a whole program cannot be encrypted using a unique stream. Indeed, in case of a jump, the stream used at the destination address must be known to keep decrypting instructions. The only way to recover it is to re-compute the entire stream from program start to the target instruction, which would be quite inefficient for long programs. Thus, it seems clear that a finer encryption granularity should be adopted.

For this purpose let us introduce basic blocks (BB), a very useful notion in compilers world. A basic block is defined as a sequence of instructions without branch (e.g., Figure 3b, the program is made of six BB). Additionally, the only entry point of a BB is its first instruction and his output is its last instruction. On in-order processor architectures, a basic block is then always executed sequentially, and can be encrypted using a unique stream cipher sequence. By definition, it is impossible for a branch to fall in the middle of a BB, so the stream cipher state never has to be reconstructed.

To encrypt a whole program, independent stream cipher sequences are generated for each basic block, using different initialization vectors (further details on this topic will be given in Section IV-D). For a processor executing such an encrypted software, branch instructions, which notify a BB change, have to trigger a reset of the stream cipher with a new IV.

The rest of this section describes more precisely the encryption process, divided into three stages. To illustrate them, code examples are given in MIPS assembly. Registers are prefixed with a dollar symbol (e.g., `$t0`, `$a1`), all instructions used with their semantic are listed in Table I. Further details can be found in the instruction set reference [29].

TABLE I
MIPS INSTRUCTIONS USED IN THIS PAPER AND THEIR SEMANTIC

Instruction	Semantic
`li $t0, value`	$\$t0 \leftarrow value$
`lw $t0, N($a0)`	$\$t0 \leftarrow MEM[\$a0 + N]$
`add $t0, $t1, $t2`	$\$t0 \leftarrow \$t1 + \$t2$
`bne $t0, $t1, dst`	if $\$t0 \neq \$t1$, jump to dst
`jump dst`	jump to dst
`jr $t0`	jump to address in $\$t0$

[1] http://opencores.org/project,tiny_aes

1) Merging Basic Blocks for Encryption: A limitation of the basic block approach to encryption is that, programs usually have a large number of small basic blocks: for MIPS programs [28], most basic blocks have between 3 and 8 instructions. Of course this metric is highly dependant on the input program and the instruction set architecture. A high number of basic blocks means that the stream cipher `init` function will have to be called very often at the execution, possibly leading to an important slowdown.

Hopefully longer sequences can be encrypted with the same stream. Indeed, the only requirement for a sequence of instructions to be encrypted with the same stream, is that there is no incoming jump somewhere other than the first instruction. There is absolutely no restriction on the number of outgoing jumps in an encrypted sequence of instructions. It slightly differs from a basic block, which cannot have more that one outgoing jump. This structure will be called an encryptable basic block for the rest of this paper and is defined as follows.

Definition IV.1. Encryptable Basic Block (EBB): A sequence of instructions which has no incoming jumps other than at its first instruction. It may contain any number of outgoing jumps.

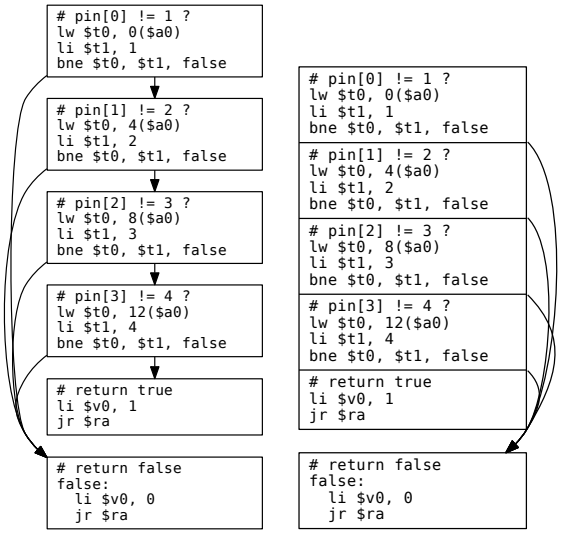
This structure is known in compiler construction as a *superblock* [30]. It is widely used to optimize programs for Very Long Instruction Word (VLIW) architectures. Some powerful techniques are available to create large superblocks: branch target expansion, loop unrolling, common subexpression detection. But an in-depth study of the optimal merging approach would brings us out of the scope of this paper.

Meanwhile, a very simple strategy is applied to merge the basic blocks. Two successive basic blocks (i.e, consecutive in the address space) can be merged for encryption if the second one has no incoming jump (can only be reached from the first one). The basic block merging for the whole program applies repeatedly this two-block merging procedure on the control flow graph, until a fixed point is reached. Even this simple approach brings performance improvements (7% in average). The merging occurs very frequently in practice, for instance while translating if-else structures. As an illustration, Figure 3 is given a very naive PIN checking algorithm, which traverses an array sequentially and returns false whenever an element does not match the expected hard-coded pin code. The control flow graph obtained using a normal compilation process generates lots of basic blocks (Figure 3b). Without merging basic blocks, it would require six different encryption sequences, one for each basic block. However this control flow graph can be fully merged into just two encryptable basic blocks as shown Figure 3c.

2) Removing Branchless Fall Through Basic Blocks: A special case to take care of for encryption correctness, is that compilers, as an optimisation, usually remove away branches going from two successive basic blocks. Indeed, the compiler assumes that when two BB are layout successors, then the first one can fall into the second one, without needing an extra jump. This behaviour has to be disabled when encrypting programs, otherwise the processor will not detect a sequence

```
bool check_pin(int *pin) {
    if (pin[0] != 1) return false;
    if (pin[1] != 2) return false;
    if (pin[2] != 3) return false;
    if (pin[3] != 4) return false;
    return true;
}
```

(a) Original C function

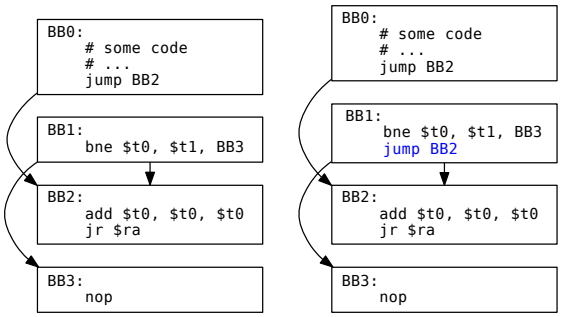


(b) Before basic block merging (c) After basic block merging

Fig. 3. A PIN checking algorithm together with its control flow graph

change and continue its execution with the wrong stream. As an example, Figure 4, BB2 has two predecessors (BB0 and BB1), and BB1 falls through BB2 without a jump. To fix this graph for encryption, an explicit direct jump to BB2 is inserted at the end of BB1 (Figure 4b).

The branch removal appears early in the LLVM compilation pipeline, so instead of modifying it directly, a late pass was implemented, that inserts back these missing branches. Then, this pass can be scheduled after the basic block merging described previously, to insert just the minimum number of branches required to fix the control flow graph.



(a) Branchless fall through (b) Fixed with a jump instruction

Fig. 4. Illustration of a branchless fall through situation (from BB1 to BB2)

D. Initialization Vector Selection Schemes

The previous section described how the control flow graph can be prepared for the encryption. To fully encrypt a graph of encryptable basic blocks, a unique encryption stream has to be generated for each one of them. The secret key being fixed and hard wired into the processor, Initialization Vectors can be used to generate distinct sequences. Unlike the secret key, IVs are public data, there is no need to keep them secret. The only requirement is that they must be unique across the whole program to guarantee security of the one-time pad encryption.

1) A Counter Mode Approach: A first possible approach is to compute IVs from the current program counter value (CTR mode). Formally, an random initialisation vector IV_0 is generated for the whole program, then to encrypt an EBB, the IV is computed as $IV = IV_0 + EBB_{addr}$, and instructions are "xored" with the corresponding stream cipher sequence (as shown Table II). IVs uniqueness is guaranteed if virtual memory isn't used, as for a given program there is only one instruction mapped to a given address.

TABLE II
BASIC BLOCK ENCRYPTION USING COUNTER MODE IVS

	Instruction	Encryption	Encryption context
			$s_0 \leftarrow \texttt{init}(IV_0 + EBB_{addr})$
i_0	`lw $t0, 0($a0)`	$i_0 \oplus r_0$	$(r_0, s_1) \leftarrow \texttt{genBits}(s_0)$
i_1	`li $t1, 1`	$i_1 \oplus r_1$	$(r_1, s_2) \leftarrow \texttt{genBits}(s_1)$
i_2	`bne $t0, $t1, dst`	$i_2 \oplus r_2$	$(r_2, s_3) \leftarrow \texttt{genBits}(s_2)$
	...	...	
i_n	`j dst`	$i_n \oplus r_n$	$(r_n, s_{n+1}) \leftarrow \texttt{genBits}(s_n)$

The benefits of this method are that there is no impact on code size, and it makes decryption dependent on current processor state. This last property can be used to build control flow integrity checking mechanisms [1].

2) Interleaving IVs in Code: Another approach is to interleave IVs within the instructions. The IV for the current EBB can be supplied using a known memory layout, as it would be done for a classic message transmission over an insecure channel. For example, the IV can be inserted at the beginning of each encryptable basic block, as shown in the example Table III.

TABLE III
BASIC BLOCK ENCRYPTION WITH INTERLEAVED IVS

	Instruction	Encryption	Encryption context
		IV	$s_0 \leftarrow \texttt{init}(IV)$
i_0	`lw $t0, 0($a0)`	$i_0 \oplus r_0$	$(r_0, s_1) \leftarrow \texttt{genBits}(s_0)$
i_1	`li $t1, 1`	$i_1 \oplus r_1$	$(r_1, s_2) \leftarrow \texttt{genBits}(s_1)$
i_2	`bne $t0, $t1, dst`	$i_2 \oplus r_2$	$(r_2, s_3) \leftarrow \texttt{genBits}(s_2)$
	...	...	
i_n	`j dst`	$i_n \oplus r_n$	$(r_n, s_{n+1}) \leftarrow \texttt{genBits}(s_n)$

Being able to use arbitrary IVs is interesting in terms of security: for instance, in the context of an output feedback mode of encryption (generalized stream cipher), provable IND-CPA [25] encryption can be achieved. This also allows the use other modes of encryption, which require uniformly random IVs, like Cipher Block Chaining mode (CBC). Another interesting application is that it makes code sharing between programs straightforward (solving one of the issues addressed in [27]), so encrypted shared libraries can be generated and dynamically linked to.

However, encryption is not anymore dependent on the address of instructions, hence code can be relocated. Unfortunately, such programs are more prone to code reuse attacks, as any EBB can be moved or called from anywhere.

3) Combining the Two Approaches: A third option is to use a combination of the two previous schemes, a random IV is interleaved within the code, and combined with the current program counter value to generate the encryption sequence. This way, the code is more resilient to code reuse attacks, while still allowing other encryption modes to be used.

E. LLVM Integration

The full code encryption process is separated in two sequential parts, a control flow restructuring part implemented in the LLVM [3] compiler framework, followed by a second part that does the encryption. The motivation for this two stage design, is to support linkage of encrypted programs, and to statically guarantee IVs uniqueness across the entire program.

Three additional passes are implemented and inserted into LLVM's MIPS code generation backend. It would be much cleaner if they could be done on LLVM intermediate representation (middle-end). Unfortunately these pass make use of basic block placement information, which are generated in early backend passes. That being said, passes are very generic and can be easily ported to other RISC targets.

The first pass does the basic block merging, the second one searches and adds jumps between fall through basic blocks and the optional third one sets up the layout for IV storage. It allocates space in the code where IVs will be stored, at this stage, memory addresses are not computed yet so these slots can be inserted without breaking the address layout. Thanks to LLVM's highly modular design, the code still benefits from late optimisation passes, including delay slot filling.

The encryption is done by a standalone program which takes as input a fully linked object file and produces the final encrypted binary that can be distributed securely to the processor. This tool reconstructs the control flow graph and internally runs a software version of the stream cipher (Trivium in our case). It is also responsible for choosing IVs for the whole program and ensures that all of them are unique.

Figure 5 illustrates the complete compile flow. For the programmer, producing an encrypted binary boils down to: 1) adding a compiler flag while compiling sources to object files, 2) encrypt the final binary. Hence the encryption can be easily integrated in a standard build system like Make.

V. HARDWARE SUPPORT

The architectural modifications required for the decryption are shown in Figure 6. Instead of providing an instruction directly from memory to the decoding logic, this value is unmasked with a stream generated by an internal cipher. Furthermore, the processor branch handling is also modified to handle IV change. Once a jump is detected, the processor executes the following steps:

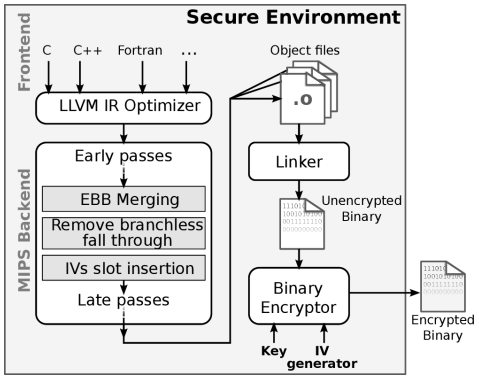


Fig. 5. Compilation flow for an encrypted program

1) Compute the IV for the branch destination address. Depending on the IV selection scheme used (discussed IV-D) either read it from instruction memory (and skip several instructions), or compute it from the current program counter value.
2) Reset the stream cipher and wait for initialization to be done.
3) Continue execution as soon as the stream is ready

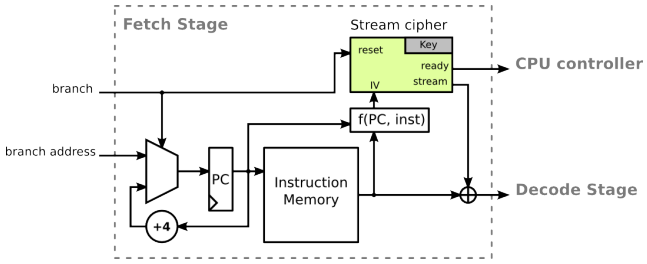


Fig. 6. Modification made to CPU's fetch stage for encryption support

A. Handling Exceptions/Interruptions

An exception is an unexpected event during program execution (division by zero, invalid memory access, interruption, ...), it is commonly handled on processors by jumping to an exception handler and putting the processor into a special mode. Once the exception is handled, the processor resumes its normal execution to the instruction where the exception was triggered.

Exceptions can still be handled while executing encrypted code. To this end, the processor must be able to restore or recompute the correct stream cipher state when returning from the exception handler. A straightforward solution would be to just save the whole stream cipher state on exception, but it would require important storage and it is not satisfying in terms of security. Indeed, anyone can compute the full encryption sequence from a given stream cipher state (see Section IV-A), so it must be kept as private as the secret key.

Instead, we remark that only the EBB address and the current offset in the EBB need to be known to restore the stream cipher state. Indeed, execution can resume at EBB start address, and instructions are skipped until the offset is reached. Then instruction execution can resume with the correct decryption stream. To continuously keep the offset in the current EBB, a register is added to track the address of the last jump. The offset in the current EBB can be computed as the difference between the current program counter value and this register.

B. Context Switching

In order to support context switching with encryption, the programmer must be able to save and restore the stream cipher state for the current process. As context switching relies on exceptions, the above solution still applies quite well, the extra register holding last jump address just has to be visible to the programmer. This way, this additional register can be saved and restored as part of the context switching routine. To this end, we overloaded the MIPS instructions `mfc0` and `mtc0`, which are dedicated to move data from and to control and status registers.

VI. Evaluation

A. Security Analysis

As the CPU machine code is encrypted using a proven IND-CPA[2] encryption scheme, it benefits from security proofs of the underlying scheme. The key is kept secret inside the processor and assumed free from observation and tampering. Then, an adversary just observing the instructions memory is equivalently viewing encrypted messages, that is, pairs of the form $(IV, \texttt{Enc}_k(IV, m))$. The security of the underlying encryption scheme guarantees that an adversary cannot learn anything about the code with static analysis.

However, encryption alone provide protection only against a very restrained attack model, in particular it does not cover:

- An adversary modifying memory (like Kuhn's attack [4]).
- Dynamic analysis of memory access patterns [31].

B. Compatibility with Software Integrity Mechanisms

Software Integrity has been ignored through this work so far. Yet, it is a real concern (see Section II). In particular when using a one-time pad encryption, ciphertexts can be easily tampered with. For instance a destination register can be changed just by "xoring" the correct field in a ciphered instruction.

Fortunately, most integrity checking mechanisms from other secure processor architectures [26] can be applied on top of our encryption method (encrypt-then-authenticate paradigm). However instruction level integrity do not seems realistic: a tag would have to be associated with each instruction, resulting in a huge code size increase. The best solution seems to authenticate data per block of fixed length, either a cache line or a buffer of instructions if the system does not have a cache. To be fully effective the integrity checking has to be done before executing a complete block of instruction, to prevent any unchecked instruction from executing.

[2]The scheme used is IND-CPA under the assumption that the function F defined by $F_k(IV) = \texttt{genBits}(\texttt{init}(k, IV))$ is a pseudorandom function [25]

C. Hardware Implementation

The hardware support described in Section V is implemented on a 32 bits MIPS [29] soft core. The processor itself is an integer only, in-order, five stage pipeline, with 32KB of read only instruction memory, and 32KB of data RAM, both are single cycle latency memory (implemented using FPGA's BRAMs). Trivium, an eStream [5] hardware profile finalist, is used as the underlying stream cipher because of its simplicity and efficiency. It has 80 bit key and IV and can be unrolled to generate up to 64 bits per cycles without increasing its circuit depth. Further unrolling can be done, but it would increase the critical path and decrease the maximum frequency.

The syntheses are done on a low cost Altera FGPA from the Cyclone V family (5CEBA4F23C7N). Table IV provides a set of synthesis results as well as maximum frequency obtained through static timing analysis. One can observe that Trivium, even unrolled, has a very small footprint and a very high maximum frequency.

For comparison, a fully pipelined 128 bit AES implementation found on OpenCores is also synthesized, it has an initial latency of 21 cycles (very close to the 18 cycles needed by Trivium x64). Although it achieves a higher throughput than the Trivium implementations, it uses far more FPGA resources. It is more than three time bigger than our base processor, and more than ten times bigger that the Trivium x64. This illustrates why stream ciphers are such good candidates for the encryption.

The complete encryption mechanism increases overall FPGA occupancy by 1.5%, and the size of the core by 26%, mainly because of Trivium circuit and little additional control hardware. Interestingly, the encryption hardware does not affect CPU's critical path, hence, the maximum frequency of the circuit is unchanged.

TABLE IV
SYNTHESIS RESULTS ON ALTERA CYCLONE V (5CEBA4F23C7N)

	Adaptative Logic Module (ALM)	f_{max}
tiny AES 128	3403 (18%)	189 MHz
Trivium_x1	148 (0.8%)	456 MHz
Trivium_x32	237 (1.2%)	360 MHz
Trivium_x64	288 (1.5%)	344 MHz
CPU base	1094 (5.9%)	108 MHz
CPU enc	1379 (7.46%)	108 MHz

D. Performance Analysis

The following evaluation methodology is used: an input program is compiled using some constant compiler flags, with and without encryption. The two resulting programs are then compared under two criteria, code size and execution time (measured in CPU cycles). The flags used are `-O3`, which optimizes the input program for execution speed, and `-Oz`, which optimizes for size.

These measurements are done for the two different IV selection schemes described in Section IV-D: IVs computed only from program counter, and IVs interleaved in code. A set of relevant programs was selected to run the above measurements. Most of them are based on open-source libraries and can be easily ported on any embedded processors. Raw results are given in Table V.

TABLE V
PERFORMANCE AND SIZE OVERHEAD RESULTS

	IVs from PC				IVs interleaved in code			
	LLVM -Oz		LLVM -O3		LLVM -Oz		LLVM -O3	
Benchmark	size	time	size	time	size	time	size	time
AES	+5 %	x2.39	+2.3 %	x1.29	27.4 %	x2.93	7.8 %	x1.41
SHA1	+6.6 %	x2.10	+6.8 %	x1.56	35.3 %	x2.52	28.6 %	x1.76
Quicksort	+9.5 %	x2.26	+11 %	x1.59	35.6 %	x2.75	30.1 %	x1.82
uECC	+7.4 %	x2.16	+7.2 %	x1.5	38.1 %	x2.61	35.7 %	x1.70
Average	+7.12 %	x2.23	+6.8 %	x1.48	34.1 %	x2.70	25.5 %	x1.67

1) Code Size Overhead: Results for size overhead are represented in Figure 7. Interestingly, even when storing IVs at the beginning of each basic block, the binary size does not increase by more than 40%. When IVs are not interleaved there is still a binary size increase due the basic block restructuring, but the observed increase does not exceed 11%.

Compiler options have a clear impact on the results. Our interpretation is that speed optimisations (enabled with option `-O3`) perform aggressive inlining and unrolling which increase basic blocks sizes, hence reduce the impact of encryption (in particular if IVs are interleaved).

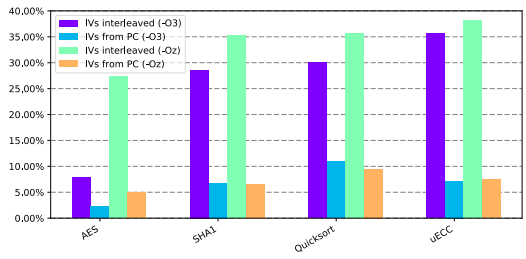


Fig. 7. Overhead factor on program size

2) Execution Time Overhead: The encryption mechanism also introduces a run-time overhead, more precisely a latency is added for each branch taken, because the stream cipher has to be initialized with a new IV. The CPU cannot be fed with new instructions while the stream is not ready. With the stream cipher used in these experiments, trivium_x64 running at CPU's clock, this latency is of 18 cycles.

The results given in Figure 8 show that the slowdown ranges from 29% up to 193%. As expected, the overhead is minimized with performance optimisations (`-O3`) and when IVs are computed from PC. In that case, the average slowdown is of 48%. When IVs are interleaved in code, the processor has to skip some (3 in our case) instructions at the beginning of each basic block, so performances are further reduced. This overhead is likely to stay reasonable, unless critical loops contain an important number of jumps.

We stress that our experimental processor does not include any cache memory, hence only the effect of the encryption is taken into account. Performances are expected to be better with an instruction cache, as the fetch on a cache miss can overlap with stream cipher initialization.

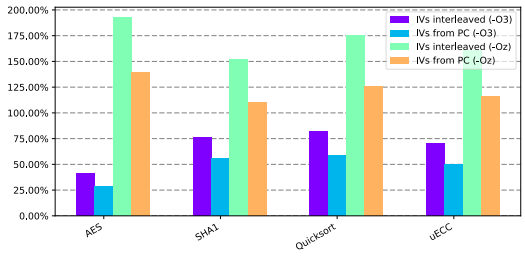


Fig. 8. Overhead factor on execution time

E. Pitfalls and Further Work

Our solution still suffers some pitfalls that we tried to identify as best as possible. First, a small hardware support must be added on-chip, which restrains the range of target systems. From a security perspective, the protection is static, and does not cover dynamic aspects, e.g., an adversary that would track and analyze memory access patterns (control flow analysis).

Further work could be done to improve performances. We saw that this kind of encryption is highly compiler dependent, this suggests that the basic block placement algorithm could be modified to maximize the basic block merging. The hardware implementation described in this paper is simple, significant speed-up is likely to be achieved with a more evolved architecture. This could be a stream cipher working at higher frequency that the CPU's clock, or to couple the stream cipher with branch prediction to begin initialization ahead.

VII. Conclusion

This paper describes an efficient method to encrypt a binary program with a stream cipher. The decryption is so fast and lightweight that it can be performed very deeply in the processor, so that plain instructions remain only in processor execution logic. The method requires slight hardware and software modification, which are implemented and evaluated on FPGA. The results are promising and opens interesting perspectives in order to improve performances and increase the range of applications.

References

[1] R. De Clercq, R. De Keulenaer, B. Coppens, B. Yang, P. Maene, K. De Bosschere, B. Preneel, B. De Sutter, and I. Verbauwhede, "Sofia: software and control flow integrity architecture," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1172–1177, IEEE, 2016.

[2] J.-L. Danger, S. Guilley, and F. Praden, "Hardware-enforced protection against software reverse-engineering based on an instruction set encoding," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, (New York, NY, USA), pp. 5:1–5:11, ACM, 2014.

[3] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[4] M. G. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP," *IEEE Transactions on Computers*, no. 10, pp. 1153–1157, 1998.

[5] M. Rogawski, "Hardware evaluation of eSTREAM candidates," 2007.

[6] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pp. 390–397, Sept 2013.

[7] A. Boileau, "Hit by a bus : Physical access attacks with firewire," 2006.

[8] A. Huang *et al.*, "Keeping secrets in hardware: The Microsoft XBox^{TM}case study," *Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2523, pp. 213–227, 2002.

[9] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *USENIX workshop on Smartcard Technology*, 1999.

[10] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic processors a survey," *Proceedings of the IEEE*, 2006.

[11] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Annual International Cryptology Conference*, pp. 1–18, Springer, 2001.

[13] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, 2015.

[14] R. Best, "Microprocessor for executing enciphered programs," Sept. 18 1979. US Patent 4,168,396.

[15] R. Best, "Crypto microprocessor for executing enciphered programs," July 14 1981. US Patent 4,278,837.

[16] D. Semiconductor, "DS5002FP secure microprocessor chip."

[17] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[18] X. Zhuang, T. Zhang, and S. Pande, "Hide: an infrastructure for efficiently protecting information leakage on the address bus," in *ACM SIGPLAN Notices*, vol. 39, pp. 72–84, ACM, 2004.

[19] G. E. Suh, C. W. O'Donnell, and S. Devadas, "AEGIS: A single-chip secure processor," *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[20] G. Duc and R. Keryell, "Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pp. 483–492, IEEE, 2006.

[21] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 183–196, IEEE Computer Society, 2007.

[22] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "ASIST: architectural support for instruction set randomization," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 981–992, ACM, 2013.

[23] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 272–280, ACM, 2003.

[24] D. McGrew, "Counter mode security: Analysis and recommendations," *Cisco Systems, November*, vol. 2, p. 4, 2002.

[25] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC Press, 2014.

[26] M. Henson and S. Taylor, "Memory encryption: a survey of existing techniques," *ACM Computing Surveys (CSUR)*, 2014.

[27] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "Making secure processors os-and performance-friendly," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 4, p. 16, 2009.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[29] I. Technologies, "MIPS32 instruction set reference."

[30] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, *et al.*, "The superblock: an effective technique for VLIW and superscalar compilation," in *Instruction-Level Parallelism*, pp. 229–248, Springer, 1993.

[31] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation.," in *NDSS*, vol. 20, p. 12, 2012.