SecSoC: A Secure System on Chip Architecture for IoT Devices

Conference Paper · June 2022

DOI: 10.1109/HOST54066.2022.9839995

2 authors:

Ayman Hroub
Birzeit University
7 PUBLICATIONS **81** CITATIONS
SEE PROFILE

Muhammad Elrabaa
King Fahd University of Petroleum and Minerals
55 PUBLICATIONS **310** CITATIONS
SEE PROFILE

SecSoC: A Secure System on Chip Architecture for IoT Devices

Ayman Hroub
Department of Electrical and Computer Engineering
Birzeit University
Birzeit, Ramallah, Plaestine
aahroub@birzeit.edu

Muhammad E. S. Elrabaa
Computer Engineering Department and Inter-Disciplinary
Research Center on Intelligent Secure Systems
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
elrabaa@kfupm.edu.sa

Abstract—**IoT technology is finding new applications every day and everywhere in our daily lives. With that, come new use cases with new challenges in terms of device and data security. One of such challenges arises from the fact that many IoT devices/nodes are no longer being deployed on owners' premises, but rather on public or private property other than the owner's. With potential physical access to the IoT node, adversaries can launch many attacks that circumvent conventional protection methods. In this paper, we propose Secure SoC (SecSoC), a secure system-on-chip architecture that mitigates such attacks. This include logical memory dump attacks, bus snooping attacks, and compromised operating systems. SecSoC relies on two main mechanisms, (1) providing security extensions to the compute engine that runs the user application without changing its instruction set, (2) adding a security management unit (SMU) that provide HW security primitives for encryption, hashing, random number generators, and secrets store (keys, certificates, etc.). SecSoC ensures that no secret or sensitive data can leave the SoC IC in plaintext. SecSoC is being implemented in Bluespec SystemVerilog. The experimental results will reveal the area, power, and cycle time overhead of these security extensions. Overall performance (total execution time) will also be evaluated using IoT benchmarks.**

Keywords—IoT security, secure processor, secure architecture, hardware security, hardware encryption, hardware decryption.

I. Introduction

As we are in the IoT era, we started to witness a numerously increasing number of IoT devices everywhere. These devices range from simple sensors that perform simple measurements, to more complex ones that perform edge computing, such as, deep learning-based autonomous vehicles. Regardless, the complexity of the IoT device, its function includes sensing data through the input/output subsystem, manipulates these data, stores, and sends the results over the Internet.

These data might be sensitive, such as, personal data, medical, military, utility metering, etc. What makes the situation even more complicated, significant part of these devices are deployed in the field and in open remote places, which makes them exposed to physical attacks. These attacks are low-budget hardware attacks. They include memory dump, bus lines snooping, modifying, reading or writing data, and injecting instructions on the bus lines, replacing hardware components by malicious ones, reprogramming memories, code injection, etc.

Many architectural solutions were proposed in both the academy and industry to secure the application's secrets in the execution environments of both general purpose-processors and domain-specific IoT processors [1-4].

ARM TrustZone [5] partitions the execution environment into secure world (private world) and normal world (public world) to provide a complete system isolation for secure code execution. However, TrustZone does not provide countermeasures against physical attacks, and hence the application's secrets can be stolen from the secure world's memory.

Intel proposed software guard extensions (SGX) [6]. The basic idea of this technology is to allow software developers to protect sensitive code and data through creating trusted regions, called enclaves.

AMD proposed secure memory encryption (SME) technology [7], which is integrated in the CPU architecture ranging from the embedded processor to the server. The encryption is performed by a dedicated hardware in the memory controller. The encryption/decryption key is managed by the AMD Secure Processor (AMD-SP).

Hong et al. [8] aimed at protecting the third-party applications' secrets on ARM-based systems where the OS is untrusted. In this work, the basic unit of protection is a function's local variables, parameters, or return value, which are called the sensitive variables. The programmer annotates these sensitive variables in the source code. Thus, at compile time, the register allocator uses registers for the sensitive variables and it never spills them to memory. These sensitive data should not leave the processor unencrypted. Software-based encryption/ decryption was used but it was not clear where the keys were stored.

Benini et al. [9] proposed an IoT multicore System-on-Chip (SoC) for edge analytics that is equipped with hardware cryptographic and computation accelerators.

The compute cores and the two accelerators communicate via shared memory. Thus, they can smoothly exchange data, which makes the encryption keys exposable.

In this work, we propose SecSoC, a novel secure system-on-chip architecture that protects the application' secrets and sensitive data. The basic strategy in this architecture is to keep all secrets of the application, such as, keys, hashes, certificates, etc. on-chip within a special purpose-processing unit called Security Management Unit (SMU). Moreover, the SMU provides hardware-based implementation of all security primitives including encryption, decryption, and hashing. In addition to that, SecSoC stores the sensitive variables of the application encrypted in the off-chip memory.

The protection takes place at both compile-time and runtime. At compile time, the programmer annotates the application's sensitive variables using compiler directives. Then, the annotated program is pre-preprocessed to resolve these annotations. After that, it is normally compiled. At runtime, in addition to the SMU, the compute cores are extended with minimal circuitry for attack detection and pulling an exception.

II. Threat Model

The main concerns of this work is securing code integrity, control flow integrity, and confidentiality and integrity of the secrets of the application running on the edge device installed in the field. Two type of secretes are to be protected: (1) security insurance secrets, such as, encryption keys, certificates, and hashes, (2) application's sensitive data.

The trusted computing base (TCB) of SecSoC architecture comprises the system on-chip hardware components. Moreover, we assume that the source of data is correct. On the other hand, SecSoC does not trust the off-chip memory, the system shared bus, the operating system and its bootloader, the application, and the physical protection of the device provided by the manufacturer. Moreover, side-channel attacks, cloud security, and network security are out of scope of this work. The attacker can perform the following attacks, (1) cold boot attack, (2) bus snooping, (3) pphysical code injection, (4) ccompromising the OS.

III. Compile-Time Protection

In the source code, the programmer, using a compiler directive, can explicitly annotate any variable holding sensitive information as a sensitive variable. Moreover, the compiler will automatically detect other sensitive variables, i.e., those derived from the explicit sensitive variables, and the programmer did not annotate them.

In Advanced Encryption Standard (AES), the cipher text's size is larger than the corresponding plaintext by multiple times. Thus, the compiler should allocate sufficient memory space for the encrypted data, e.g., using arrays or vectors. Moreover, the compiler should generate sufficient number of load/store instructions to load/store the encrypted data. In this paper, and for simplicity, we will assume a 128-bit key.

Thus, the compiler generates a chunk of consecutive load/store instructions to load/store an encrypted data item. Let us call this chunk of instructions the load/store block or the sensitive block. To mark the beginning and the end of this block, the compiler inserts a special affectless instruction before and after each load/store block. Let us call these instructions a *Begin* and *End*. In RISC V, this instruction can be any instruction that does not change the programmer-visible state. It only advances the program counter (PC) and increments any applicable performance counter. For example, it can be *ADDI x0, x0, imm*, where *x0* is hardwired to zero, and any instruction tries to modify it will be discarded. *ADDI x0, x0, imm* can be used to encode *Begin*, and *ADDI x0, x1, imm* can be used to encode *End*. Moreover, each load/store block is given a unique serial number that can be encoded using the immediate value. This serial number will help in code integrity insurance. Fig. 1 shows a C++ code snippet with a sensitive variable. Fig. 2 shows the corresponding RISC V assembly code.

IV. SecSoC Architecture

A. SecSoC Overview

Fig. 3 shows the high level SecSoC architecture. It comprises the following components that are interconnected via a shared system bus:

1. SecSoC chip, which consists of, (1) RISC V-based compute engine, which executes the user's application. (2) Security Management Unit (SMU), which is a special purpose processor. It stores the security insurance secrets, i.e., the processors' secret keys, certificates, hashes, etc. in a read-only nonvolatile memory. Moreover, it provides hardware accelerated hardware implementation of the security primitives, namely, encryption, decryption, and hashing.
2. Main memory, which is usually a DRAM. It stores the application's code and data.
3. Nonvolatile memory to store the application's persistent data, e.g., learned models.
4. I/O devices, i.e., the various I/O devices and their controllers in the IoT node.

```
#sensitive //compiler directive
int b = 10; // b is sensitive variable
int c; // c is a non-sensitive variable
int a = b + c; //'a' becomes sensitive variable
b++;
```

Fig. 1: C++ Code Snippet with a Sensitive Variable

B. *RISC V-based Compute Engine*

The SecSoC's compute engine can be a single or a multi-core processor. Moreover, it can include compute accelerators. In this paper and for simplicity, we assume a 5-stage pipelined single core RISC V-based processor, and hence we will use the terms compute core and compute engine interchangeably in this paper. This processor operates in two modes, namely, the *normal* and the *secure* mode.

In addition to its original functional components, the compute core has been extended with some minimal circuitry to perform security monitoring and control. This extension includes register file extension, flag bits, control signals, and control logic.

1) *Register File Extensions*

The register content is written back to memory either by an explicit store instruction in the user's application, or upon a context saving in the case of a function call, thread switching, interrupt, etc. Whenever a sensitive register needs to be stored to memory, the encrypted version of this register is stored. On the other hand, whenever a sensitive register is loaded from memory, it has to be decrypted first by the SMU. Encryption and decryption are blocking operations, i.e., the processor stalls until the encryption/decryption operation completes. SecSoC register has been extended with the following:

1. Shadow Register File (SRF), it stores the decrypted values of the sensitive variables.
2. Extended Register File (ERF) is an extension to the GPRF. It stores the encrypted value of the sensitive variable that is corresponding to the decrypted one in the SRF.
3. Security bit vector (S): each register of the GPRF is extended by a security bit. It indicates whether the corresponding register holds a sensitive variable.
4. Modified bit vector (M): each register of the GPRF is further extended by a modified bit. It indicates whether the corresponding register's content has been modified or not. This is useful when this register is saved upon a context switch. If it is not modified, then the stored encrypted value in the GPRF is saved as is without a need for an encryption.
5. The sensitive register can be loaded/stored only and only in the *secure* mode. Otherwise, the processor throws an exception.

```
# Assume the following register allocation
#x1 <-- a, x5 <-- b, x6 <-- c
Begin 0# beginning of a load block #0
lw x5, 40(x10)#load the encrypted variable b
lw x5, 44(x10)
lw x5, 48(x10)
lw x5, 52(x10)
End 0# end of a load block #0
#Decrypt b
lw x6,100(10) # load non-sensitive c
add x1, x5, x6 # a = b + c;
#Encrypt a
Begin 1# beginning of a store block #1
sw x1,60(x11)#store the encrypted variable a
sw x1,64(x11)
sw x1,68(x11)
sw x1,672(x11)
End 1# end of a store block #1
andi x5, x5, 1 # b++;
#Encrypt b
Begin 2# beginning of a store block #2
sw x5, 40(x10)#store the encrypted variable b
sw x5, 44(x10)
sw x5, 48(x10)
sw x5, 52(x10)
End 2# end of a store block #2
```

Fig. 2: RISC V Assembly Code Corresponding to the C++

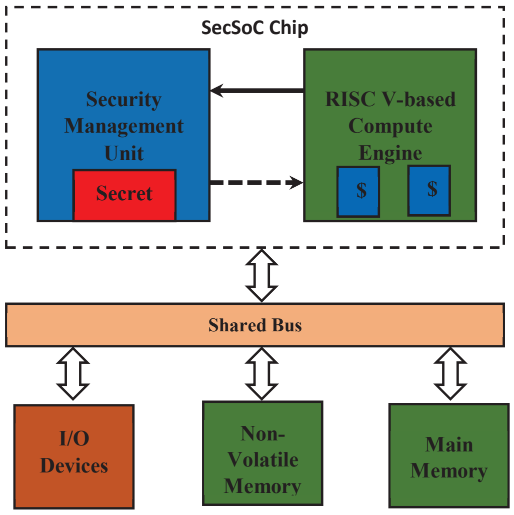

Fig. 3: Overview of SecSoC Architecture

When a register is loaded from memory with sensitive data, i.e., the processor works in the *secure* mode, the encrypted version of the sensitive data is saved in the GPRF and its extended ERF, the corresponding security bit is set, and the corresponding modified bit is initially cleared. Moreover, a copy of this encrypted version is sent to the SMU for decryption. When the decryption completes, the decrypted value is stored in the corresponding SRF register. On the other hand, when a register is loaded with non-sensitive data, i.e., the processor works in the *normal* mode, the loaded data is stored in the original GPRF, the corresponding security bit is cleared, and the modified bit value has no effect.

SRF, which stores the decrypted version of the sensitive variables, is not directly visible to the programmer. If the security bit of the source register is set, then the register value is fetched from the SRF. Otherwise, it is fetched from the GPRF.

2) Decode Unit Extension

The decode unit security extensions include the following. When the decode unit detects an attack attempt, the processor throws an exception.

1. When the *Begin* instruction is encountered by the decode unit, the processor is switched to the *secure* mode.
2. The decode unit checks the correctness of the serial number of the *Begin* instruction.
3. The decode unit checks that the instruction count in the sensitive block is correct.
4. The decode unit forwards a copy of each instruction in the sensitive block to the SMU for hash computing of the block.
5. The decode unit verifies that all instructions in the sensitive block are either all load or all store.
6. The compute core stalls until hashing, encryption or decryption of the sensitive block is completed
7. The decode unit verifies that each *Begin has an End* with the same serial number.

C. Security Management Unit (SMU)

SMU is a special purpose processor responsible for security management in SecSoC. It stores the application's secrets in electrically erasable read-only nonvolatile memory at configuration time, whereas the master secret key is burned during fabrication at the foundry. Moreover, the SMU stores the hash values of the application's sensitive blocks.

In addition to storing the application's secrets, SMU efficiently implements all security primitives. Thus, when the compute core encounters a sensitive block, it performs the following, (1) sending all instructions of this block to the SMU for hash computing, (2) sending a copy of data within the load/store block to the SMU for decryption or encryption.

V. Implementation

Bluespec open source 5-stage pipelined Flute RISC-V core [10] has been selected to be the base compute core in the proposed SoC. This is because it is ideal for low-edge applications and embedded systems. We are modifying this core to add the architectural security extensions, without changing the instruction set architecture. Moreover, we are implementing an AES-based SMU.

VI. Conclusion

In this work-in-progress paper, we proposed SecSoC, a novel RISC V-based SoC secure architecture to protect the application's secrets and sensitive data in the IoT nodes. SecSoC does not require changing the instruction set, the compiler, and the operating system. It allows the programmer to determine which variables are sensitive in the source code. Then, a preprocessor processes these annotations before compilation. Currently, we are implementing SecSoC in Bluespec SystemVerilog to verify its functionality, and assess the performance and cost overhead of the security extensions.

Acknowledgment

The authors would like to acknowledge King Fahd University of Petroleum and Minerals and Birzeit University for supporting this research.

References

[1] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-Assisted Secure Execution on ARM Processors," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 72-90.

[2] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," presented at the Proceedings of the 20th USENIX conference on Security, San Francisco, CA, 2011.

[3] J. P. McGregor and R. B. Lee, "Protecting cryptographic keys and computations via virtual secure coprocessing," *SIGARCH Comput. Archit. News,* vol. 33, no. 1, pp. 16–26, 2005.

[4] P. Colp *et al.*, "Protecting Data on Smartphones and Tablets from Memory Attacks," presented at the Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, Turkey, 2015. Available: https://doi.org/10.1145/2694344.2694380

[5] ARM, "ARM Security Technology, Building a Secure System using TrustZone Technology," *White Paper,* 2005-2009.

[6] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Trans. Comput. Syst.,* vol. 33, no. 3, p. Article 8, 2015.

[7] J. P. David Kaplan, Tom Woller "AMD MEMORY ENCRYPTION " *White Paper,* 2016.

[8] M. H. Yun and L. Zhong, "Ginseng: Keeping Secrets in Registers When You Distrust the Operating System," in *NDSS*, 2019.

[9] F. Conti *et al.*, "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 64, no. 9, pp. 2481-2494, 2017.

[10] (2021). *Open-source RISC-V CPUs from Bluespec, Inc.* Available: https://github.com/bluespec/Piccolo