Intellectual Property Protection for Embedded Sensor Nodes

Michael Gora, Eric Simpson, and Patrick Schaumont

Virginia Polytechnic Institute and State University,
Secure Embedded Systems Group,
302 Whittemore Hall (0111), Blacksburg VA 24061, USA
{gora,esimpson,schaum}@vt.edu
http://www.ece.vt.edu/schaum/research.html

Abstract. Embedded Sensor Networks are deeply immersed in their environment, and are difficult to protect from abuse or theft. Yet the software contained within these remote sensors often represents years of development, and requires adequate protection. We present a software based solution for the Texas Instruments C5509A DSP processor which uses object-code encryption and public-key key exchange with a server. The scheme is tightly integrated into the tool flow of the DSP processor and compatible with existing embedded processor design flows. We present performance and overhead metrics of the encryption algorithms and the security protocols. We also describe the limitations of the solution that originate from its software-only, backwards-compatible nature.

1 Introduction

Securing intellectual property in embedded applications is an ever growing concern for developers. These concerns are even more prevalent when such applications are deployed in unsecure and hostile environments as is often the case with sensor networks. Code utilized on such nodes can represent a major investment on the part of the developer, yet the code is often left unprotected. A common fear is that such an unsecured product, discarded or stolen, appears on the black market where it can be obtained by a competitor. Code stored in plain text could easily be copied and deployed on a competing platform damaging the original developers market position. Even worse in the case of critically important networks, code could be reverse engineered to aid in the disruption of service or theft of sensitive data. Solutions lend themselves to hardware based approaches for securing newly developed systems [1]. However, this leaves a great deal of older systems that run on a legacy platform vulnerable. Rather than opting for costly hardware retrofits for such systems, a software approach may extend the platforms useful application life.

Our work presents such a solution for securing firmware-based intellectual property (FIP) on embedded sensor nodes. The solution is geared to be compatible with the existing design flow for the Texas Instruments C5509A DSP (C55). Figure 1 illustrates the two parts of our solution. First, tight integration

M. Berekovic, N. Dimopoulos, and S. Wong (Eds.): SAMOS 2008, LNCS 5114, pp. 289–298, 2008.

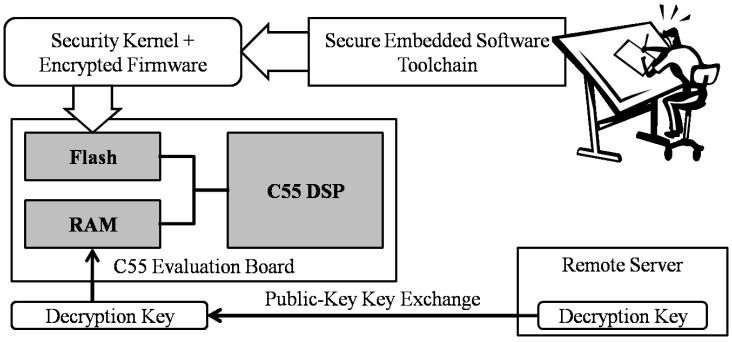

Fig. 1. IP security schema overview

of IP encryption and the software tool-chain provides a novel and streamlined method for the protection of firmware. This is extended with a security kernel which provides a platform for the authentication and decryption of secured code at boot.

Second, the security kernel negotiates a firmware based intellectual property (FIP) decryption key from a key server at startup. The use of a key server is required as the C55 does not posses any secure nonvolatile memory. Under the generic nature of the implementation it can not be assumed there is hardware present that does. However, the introduction of a key server requires an authentication procedure, in order to avoid man-in-the-middle attacks. This is further addressed in Section 6. Here, we assume that the sensor node can be reliably authenticated by the key server. We use a public-key exchange protocol based on an Elliptic Curve Diffie-Hellman (ECDH) protocol. The approach of storing the key off of the sensor node prevents the simple decryption of the FIP by reverse engineering of the firmware. The firmware key can only be obtained by booting the node and completing the key-exchange. The retrieved FIP key is utilized internally on the processor to decrypt the firmware. As the key exchange and decryption can occur only at boot there is no required runtime overhead. Once the ECDH key exchange has completed, the firmware decryption service has a footprint of only 7.3 Kbyte. To our knowledge this is the first published result of a complete end to end implementation of a firmware encryption scheme combined with an ECC public-key exchange on a DSP. We have verified our approach by building an end-to-end prototype of the entire system, including sensor node and key exchange server.

The paper is organized as follows. Section 2 outlines the assumptions that shaped our design decisions. Section 3 covers the methodology for the encryption and decryption of the firmware object code. Section 4 presents the implementation details of ECDH on the C55. The performance of both ECDH and firmware encryption are reported in Section 5 and compared to other platforms. Section 6 analyzes strengths and weaknesses of our solution while Section 7 summarizes the project and indicates areas for future work.

2 Constraints

Our primary focus is the creation of a software-only protection mechanism to secure intellectual property in firmware on a Texas Instruments C5509A DSP, a 16-bit processor. In addition, maximal flexibility is ensured in development, by creating portable code in C, and by integrating the firmware encryption flow in the C55s software development environment, Code Composer Studio 3.1 (CCS). All additions to the tool-chain to facilitate this are also written in portable C code for the GNU Compiler Collection 4.2.0. All encryption schemes are developed with a minimum of 128bit AES secret key security or equivalent [3] as specified by the NSA guidelines [2].

Given the constraints outlined above, we opted for a combination of firmware encryption with a remote key-exchange. Indeed, as this is only a software based solution the addition of a specific hardware component to securely store or generate this key is not an option. We therefore use a public-key key exchange mechanism to retrieve the firmware decryption key. The resulting arrangement is divided into two distinct components, IP encryption/decryption, and key transmission.

3 IP Encryption and Decryption

3.1 Identification and Encryption

Identification and encryption are a tightly coupled step in our implementation. The final binary requires plain text code sections. These perform such tasks as key exchange, authentication, firmware decryption, and traditional boot up tasks. Identification of the sensitive IP and non-critical code sections is accomplished during development through the built in code section pragmas made available by CCS. The net effect of singling out only the critical IP allows code to be selectively encrypted allowing for smaller decryption times.

Encryption of the selected code sections occurs after the compilation and linking of the design results in a complete binary and is a post processing step. As we have adopted the strategy of allowing individual sections of firmware to be encrypted it is necessary that these sections are logical entities handled by the DSP compiler and linker. As such we obtain tight integration between firmware encryption and firmware production. A development tool included with CCS, OFD55, provides detailed information on each section contained in a binary file, including the size and offset of each. Figure 2 demonstrates how a compiled binary file resulting from CCS is encrypted.

The Object Encryptor (OE) is a utility we developed that encypts a plain text binary. The developer can choose what sections in the binary should be encrypted by providing a sections file. The sections file only contains the names of the identified sections to be encrypted. The offset and length of the sections are provided by the OFD55 utility from the CCS tool chain (OFD file). The OE next uses a designer-provided key (Key file) and an arbitrarily generated nonce to encrypt the designated code sections. For additional security the OE allows the use of different Keys and nonce to be used on different code section. This

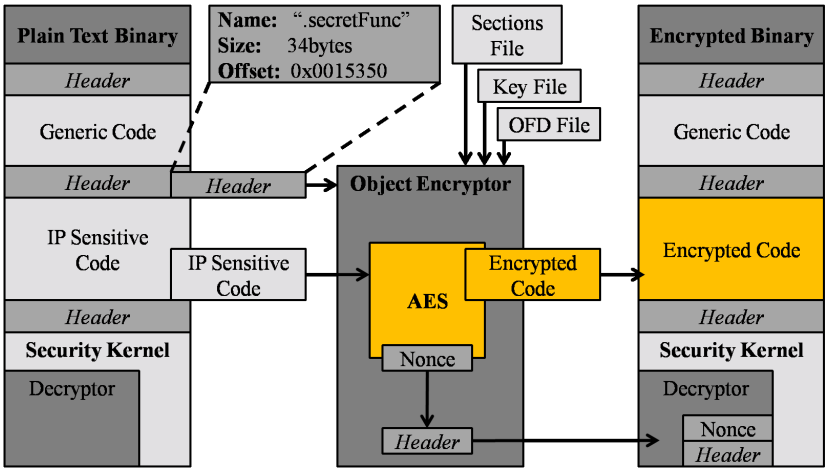

Fig. 2. Object Encryption

provides a greater flexibility in key and IP management by allowing the developer to specify different key management policies for each section. Generating the encrypted key stream is accomplished through AES in Counter Mode [11]. An AES key length of 128 bits is used as to satisfy the requirements for secret level clearance specified by the NSA standard [2]. The AES Counter mode allows the use of a key-stream in blocks of 16 bits, as is needed for the native word length in the C55 processor. At the same time, it also avoids the requirement that code sections need to be a multiple of 128 bits.

Besides the encryption of firmware sections, the OE also creates an additional data section in the resulting encrypted binary. Space for this data section is allotted in the security kernel. The plain text data section holds the offset, size, and nonce information for each IP sensitive code section that was encrypted. This data section is used by the Security Kernel at boot time to locate encrypted firmware and decrypt it into executable object code. After the OE concludes the resulting binary will contain both encrypted and plain text code sections. Any standard methods of deploying the binary may be then used.

3.2 Decryption in the Security Kernel

Decryption may be handled in two ways, a one time cost to decrypt all encrypted firmware at boot or a distributed run time cost to decrypt individual sections when needed. Regardless, decryption follows the same general methodology and should only be performed on internal DSP memory. At any time unprotected code only exists in the C55, where it is assumed to be secure, as the abundance of fast and tightly controlled memory alleviates the necessity of utilizing chip ram. JTAG and other security concerns are further addressed in Section 6 of this paper.

The actual decryption routines in the security kernel are always present in plain text. However except for software integrity issues, which are addressed in Section 6, this is not of concern. During decryption the section information stored in the security kernel by the OE is used to set up encrypted sections for decryption. The only missing information for decryption is the 128bit key, which must be brought from a secure external source.

3.3 C55 Design Flow

One of the primary goals of this work is to provide an IP encryption solution that is easily utilized across a wide series of potential target applications. As such it is necessary to consider the development suite, design flow, and deployment for a typical C55 implementation. A generic implementation containing assembly and C code is compiled or assembled before being placed as dictated by the memory map. These placed code sections are then linked appropriately before being written to a binary output file. Any post processing is then performed before the binary is flashed to the C55 and executed. The only additions to the design are the addition of the security kernel which is developed with only low level C code and assembly functions as to have as minimal impact. Inclusion of the OE is the only addition to the flow and will generate encrypted code sections as identified by the designer. No other design alterations are required after these initial steps. The only remaining step is to change the boot vector of the C55 to run the security kernel upon processor reset.

4 Key Transmission

4.1 Overview

Communication between the C55 and the key server occurs over an open unsecure channel in our implementation. As such the establishment of a secure channel is required before any key exchange may occur. A public key protocol such as Diffie-Hellman is perfectly suited to such a task. Diffie-Hellman (DH) is a well known mechanism for public key cryptography across many different platforms. We utilize Diffie-Hellman over Elliptic Curves, which is well suited for embedded applications. Indeed, an implementation of Elliptic Curves over a 256 bit prime field provides equivalent security compared to an RSA key of 3072 bits, which corresponds to an 128-bit secret key. Thus, a 256-bit prime field provides secret-level security according to the NSA standard [2].

While other highly portable C code implementations of ECDH exist (such as LibTomCrypt [5]) these are not suited to deployment on the C55. An embedded implementation of ECDH, TinyECC [6], requires the use of TinyOS and the nesC compiler in the tool chain. We opted against using TinyECC as to maintain compatibility with the existing design path. After careful consideration it was deemed necessary to implement ECDH from the ground up. This includes the extended precision finite field (GF) arithmetic necessary to implement EC, the math functions to implement an EC point multiplication and the DH protocol that relies on EC point multiplications to derive public and secret keys.

4.2 Diffie-Hellman Protocol

ECDH is a well known exchange protocol and as such only its specific implementation will be covered. For the purpose of this implementation only one ECDH exchange will occur during boot of the target system. During this single cycle a public key is derived and transmitted between the DSP and server systems. Each public key is used to derive a 128 bit private key that can be used to transmit the IP decryption key through an AES block cipher as summarized in Fig. 3.

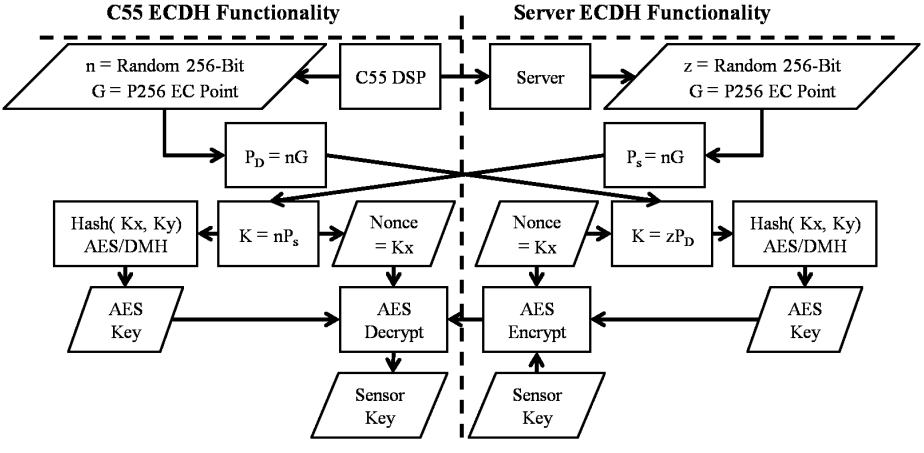

Fig. 3. Diffie-Hellman public key exchange and AES key derivation

Deriving a public key requires that both DSP and server sides of the scheme use the same base point. We use the IEEE standard for the 256 GF(p) field [7]. A single EC scalar multiplication creates the desired public key whose bit stream including its degree is transmitted over the insecure channel. Each public key once received by the opposite platform requires an additional scalar multiplication against that platforms previously determined number. The result is an identical private key on both server and DSP sides of the implementation. The private key is represented as a point (X,Y) of two 256 bit fields. The same AES 128 bit implementation used to decrypt the IP sensitive sections of code is utilized to transmit the key. To generate the key it is necessary to reduce the 512 bits of private key into a 128 bit AES key. This compression is obtained through a Davies Meyer hash implementation.

4.3 Elliptic Curve Arithmetic and Finite Field

Elliptic Curve arithmetic is built on top of modular arithmetic, and creates public and secret keys by multiplying a point on an elliptic curve by a scalar value. By default, points are represent in the affine (X,Y) coordinate system. For efficiency reasons, embedded system implementations internally apply projective

format (X,Y,Z) for points, and include conversions from/to affine to projective format as needed. The IEEE standard [7] provides generic implementations for addition, subtraction, doubling, conversion between affine and projective, and scalar multiplication.

The modular arithmetic for point operations are based on finite field arithmetic of either the prime (GF(p)) or binary (GF(2P)) type. We used the GF(p) scheme, in part because an easily accessible open-source implementation was available that could be used as a golden reference [5]. Field length for the finite arithmetic is another system parameter to choose. According to Table 1, a secret equivalent protection for a 128-bit private key required us to use a 256-bit prime field. GF(p) requires an efficient embedded implementation of multi-precision arithmetic operations. However for the lowest level of these such as addition and multiplication there is no easy access to the carry bits and leads to large and complex implementation. To combat this problem addition, subtraction, shift operations, and comparisons are written in assembly.

5 Results

5.1 Demonstrator Components

The demonstrator hardware contains a Spectrum Digital C55 Development System and a server running the key-server functionality. The communications link between the C55 board and the server is based on USB, but easily replaceable with other technologies. The software on the C55 DSP board includes a USB communications library, the security kernel containing the ECDH protocol and the Object Decryptor, and finally an encrypted C55 application. The server server contains a similar USB library, a matching ECDH protocol and the secret key that can decrypt the object code. Software for the C55 kit is developed in CCS on a development system, which also contains the Object Code Encryptor. Once the application is generated and encrypted, it is downloaded into the Flash memory of the C55 board. The key used to encrypt the application is installed on the Server. Next, the C55 board can be booted and will go through a complete key exchange and application decryption sequence.

5.2 Encryption Performance for the C55

Through testing on our demonstrator components we obtained an average performance of approximately 21 million cycles or 105 milliseconds for one ECDH exchange on the C55 processor. This value is obtained by performing several different key exchanges with different 256-bit scalar values. We then compared this performance with several different published implementations. The comparison is done in seconds normalized over the operating frequency of the platform. The results of this comparison are captured in Fig. 4. This demonstrates that our implementation on a 16-bit platform compares favorable to some of the published 32-bit platforms. We also evaluated the symmetric-key encryption performance

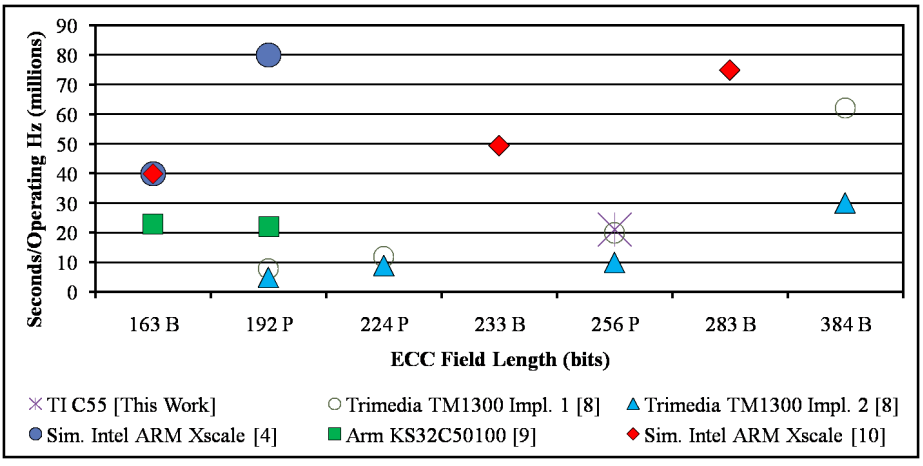

Fig. 4. ECDH speed comparison

on the C55 and evaluated that to be 2023 cycles per 128 bits. We can also observe that the symmetric-key encryption speed is 3 orders of magnitude faster then public-key encryption.

For the complete protocol, we evaluated that the ECDH handshake and subsequent decryption of 128 Kbytes of firmware takes about 40 million cycles on the C55. Since ECDH consumes 20 million cycles, it thus takes roughly the same amount of time to decrypt a block of 128 Kilobytes of code as it takes to perform two ECC point multiplications (one complete ECDH handshake). The complete on-chip memory space of the C55A contains 256 Kilobyte, and the security kernel will never decrypt more than this during boot. Hence, it would be necessary to optimize the current symmetric-key decryption speed before improving ECDH protocol implementation. The memory footprint for the security kernel is approximately 17.3Kb or merely 6.7% of available onboard memory for the C55. This is broken up between two sections AES and ECDH which respectively have footprints of 7.1Kb and 13.3KB. It should be noted that ECDH also utilizes the Rijndael algorithm to perform a Davies Meyer hash on the private key value to generate an AES transmission key. This represents the 3.1 K byte discrepancy in size between the two footprints. Upon retrieving the firmware key ECDH may be discarded leaving a run time footprint of 7.1Kb for AES decryption, or 2.7% of available memory.

6 Security Analysis

In this section, we discuss the challenges of implementing a firmware protection technique using only software techniques. We are interested in securing off-chip object code. Once the off-chip object code is loaded from nonvolatile memory onto the processor and decrypted, it is no longer protected. Hence, we assume

that the C55 processor package itself can be protected from external inspection or tampering. This requires additional precautions, such as security measures for the chip JTAG interfaces [12]. Securing such vulnerabilities on an existing system is not reasonably done in a generic implementation and if possible would require tight integration with the end application. We also assume that the encrypted firmware itself can be trusted. Any vulnerability in this code such as buffer overflows or unchecked data access would lead to an additional security breach.

6.1 System Authentication and Integrity

System authentication and integrity are of crucial concern to a software only solution. Due to the nature of the C55 and its lack of secured nonvolatile memory these issues present themselves outside the scope of such a solution. For the purposes of this paper we thus assume that the end-user of the system is able to guarantee the integrity of the security kernel. This is required to thwart an attack that would compromise the platform by code injection, or through hardware emulation. Booting with a compromised security kernel or in software that was running on an emulated system would leave the decrypted code sections vulnerable. Solutions that provide security kernel integrity can either rely on physical protection, or else use a hardware-based hashing facility [13]. Processors with on-chip non-volatile memory are able to store the security kernel on-chip [14]. For a RAM-only processor such as the C55, an add-on SHA-1 hardware module with a write-only hashing facility can be used as a building block for integrity verification. A secure hash can be combined with an encryption key into a keyed-Hash Message Authentication Code (HMAC). This can be used to both verify the integrity and the authenticity of the node simultaneously [15]. A failure to respond correctly to such a response would result in the denial of a decryption key as per a key management scheme. Finally, we emphasize that the limitations are all originating from the desire to support firmware protection on legacy platforms. Part of our efforts has been to identify exactly those risks mentioned above, and to analyze possible countermeasures.

7 Conclusions

We have presented a complete demonstrator for firmware code encryption on embedded sensor nodes. Our results show that such a mechanism can be systematically integrated into a TI C55 software production flow, and that the resulting overhead on system resources is minimal. We have achieved software-only code security by storing secrets off-platform in a key-server. While this may not be an option for all embedded sensor situations, it did fit the purpose of our project. The code encryption flow is presently being adopted by our industrial partner. We are considering further improvements on the protocol and its implementation, including hardware authentication of the C55 platform to the server and the protection of C55 interfaces and debug ports which could affect the sensor node at runtime.

References

1. TCG Mobile Trusted Module Specification v 1.0 (June 2007), http://www.trustedcomputinggroup.com
2. CNSS: National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information. ICNSS Policy No. 15 Fact Sheet No. 1, Ft. Mead (2003)
3. Giry, D.: Recommended Cryptograph Keylength, http://www.keylength.com
4. Branovic, I., Giorgi, R., Martinelli, E.: A Workload Characterization of Elliptic Curve Cryptography Methods in Embedded Environments. In: ACM SIGARCH workshop on Memory Performance, pp. 27–34. ACM, New York (2003)
5. LibTomCrypt, http://libtom.org
6. TinyECC, ECC for Sensor Networks, http://discovery.csc.ncsu.edu/software/TinyECC/
7. Microprocessor and Microcomputer Standards Committee of the IEEE Computer Society: IEEE Standard Specifications for Public Key Cryptography. IEEE-SA Standards Board, New York (2000)
8. Hu, Y., Li, Q., Kuo, C.-C.: Efficient Implementation of Elliptic Curve Cryptography (ECC) on VLIW-Micro- Architecture Media Processor. In: 2nd IEEE ICME, pp. 181–184. IEEE Press, New York (2004)
9. Wollinger, T., Pelzl, J., Wittelsberger, V., Paar, C.: Elliptic and Hyperelliptic Curves on Embedded P. In: 3rd ACM TCES, pp. 509–533. ACM, New York (2004)
10. Bartolini, S., Branovic, I., Giorgi, R., Martinelli, E.: A Performance Evaluation of ARM ISA extensions for Elliptic Curve Cryptography Over Binary Finite Fields. In: 16th IEEE CAHPC, pp. 238–245. IEEE Press, New York (2004)
11. Ferguson, N., Schneier, B.: Practical Cryptography. Wiley Publishing, Inc., Indianapolis (2004)
12. Buskey, R.F., Frosik, B.B.: Protected JTAG. In: IEEE Parallel Processing Workshop, p. 8. IEEE Press, New York (2006)
13. Dallas Semiconductor: White Paper 8: 1-Wire SHA-1 Overview (September 2002), http://www.maxim-ic.com/
14. Suh, G., O'Donnel, C., Sachdev, I., Devadas, S.: Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In: 32nd IEEE ISCA, pp. 25–36. IEEE Press, New York (2005)
15. Bellare, M., Canettiy, R., Krawczykz, H.: Message Authentication using Hash Functions: The HMAC Construction. In: 2nd CryptoBytes, RSA Laboratories, Bedford, vol. 1, pp. 25–36 (1996)