Secure Execution with Components from Untrusted Foundries

Rahul Simha Bhagirath Narahari
Department of Computer Science
The George Washington University
Washington, DC 20052
{*simha, narahari*}*@gwu.edu*

Joseph Zambreno Alok Choudhary
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208
{*zambro1, choudhar*}*@eecs.northwestern.edu*

Abstract

As the cost of manufacturing microprocessors continues to rise, chip makers have begun accelerating the export of fabrication capabilities to overseas locations. This trend may one day result in an adversarial relationship between system builders and chip producers. In this scenario the foundries may hide additional circuitry to not only compromise operation or leak keys, but also to enable software-triggered attacks and to facilitate reverse engineering. At the same time, users desire modern computing platforms with current software, along with the ability to run both secure and non-secure applications at will. In this paper we describe an approach to address this untrusted foundry *problem. We demonstrate a method by which system builders can design trusted systems using untrusted chips from these remote foundries. The key principle behind our approach is that we use a multi-layer encryption protocol alongside a dual-processor architecture. Using this approach any information leaked to an attacker will be of little value.*

1. Introduction

Computing systems today are manufactured in a complex marketplace that features multiple chains of consumers and producers. In this marketplace, for example, a hardware producer such as Motorola sells chips but is itself a consumer when it contracts out the actual physical production of its design to a foundry. Similarly, producers of end-user devices such as cellphones are consumers of the chips that go into their devices. A recent development in this marketplace is that as the chip technologies pursue ever smaller dimensions, foundries have become extremely expensive to build and maintain [3]. As a result, the economics of this marketplace have led to the current situation where most foundries are concentrated outside the United States and Europe. This situation raises the possibility that such external foundries, which are not subject to the tight security customary in government facilities, may be compromised. In the case of a microprocessor, the most harmful type of compromise arises when an adversary exploits the sheer complexity of modern circuits to insert a small, hard-to-detect, so-called Trojan circuit into the chip.

What advantage might an adversary secure with such a hidden Trojan circuit? We describe possible attacks on end products such as radios or computers based on such chips even when the software has been subject to rigorous standards of security in its design and operation. In the case that the end product is a computer, such a high standard usually means the entire execution occurs in encrypted form [7] - software is completely encrypted in main memory and is only decrypted at the time of execution in the processor. A hidden circuit inside the processor can easily thwart this approach by seizing control of the processor at an opportune moment and writing out decryption keys onto peripherals. This attack is commonly called a leakage attack. An even more rudimentary attack is to simply halt the processor at the worst possible moment, at a critical or random time long after the processor has been in use. Hidden circuits can also be set up to scan for electromagnetic signals

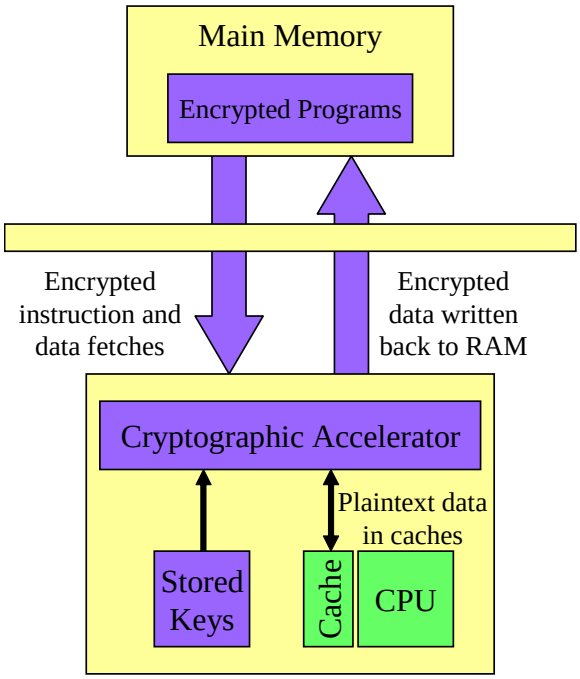

Figure 1. Encrypted execution and data platforms

so that a processor can be shutdown when provided the right external cue. Another useful attack from the point of view of the adversary is to use such circuits to facilitate reverse engineering of system design. In this sense, the problem we have described is not simply limited to military or government applications but is also of interest to commercial entities concerned with guarding their intellectual property.

Networking hardware may be expecially vulnerable to this sort of attack, as a compromised chip could be used to facilitate remote exploits. For example, the chip manufacturing process could be subverted to produce hardware firewalls that grant complete external access to the network as a response to some trigger condition. This external activation could be as simple as a packet sent from a predetermined network location or something more complex like a key encoded as a series of requests to different ports. Such a vulnerability would be very difficult to pinpoint, especially if an attack is only triggered on rare occasions.

This paper focuses on the design of systems using chips from untrusted foundries. We do not consider the equally important problem of detecting such circuits by subjecting chips to external black-box tests or imaging techniques. Our primary goal is to detect the moment such a hidden circuit ”makes its move” and then to raise an alarm to the user. We assume that the system is itself built in a trusted location and that no collusion exists between foundries and system developers (such as board makers or software developers). Furthermore, our solution is targeted at a very high level of security and therefore we assume encrypted execution, as will be explained in the next section. Our approach to the untrusted foundry problem can be summarized as follows. We utilize a multi-layer encryption protocol, a two-processor architecture, and a trusted tool-chain under the supervision of architects and developers of the targeted secure applications. The core of our approach is the way in which the two processors are used: one is configured as a gateway, while the other is configured as an execution engine.

The remainder of this paper is organized as follows. In Section 2, we provide a brief summary of related work in the area of hardware support for security. In the following sections, we describe our proposed approach in more detail, and then discuss our future plans for evaluating its effectiveness.

2. Related Work

Many previous secure architectures have focused on providing an encrypted execution environment. Figure 1 shows an encrypted program residing in main memory where the cryptographic keys are in the processor. When instructions or data are loaded into the processor, they are decrypted inside the processor. Likewise data that is written back to main memory is encrypted using these same stored keys. The purpose of the encryption is to prevent information leakage over the bus. Such leakage can occur when the product is captured by an attacker with access to a sophisticate laboratory in which the attack can snoop on the bus or directly manipulate the bus itself.

Encrypted execution has been studied by various researchers. In [7], an architecture for tamper-resistant software is proposed, based on an eXecute-Only Memory (XOM) model that allows instructions stored in memory to be executed but not manipulated. Specialized hardware is used to accelerate cryptographic functionality needed to protect data and instructions on a per-process basis. Much of the recent research in this field has focused on improving the overhead of the required cryptographic computation in the memory fetch path - examples include the architectural optimizations proposed in [11, 16].

Pande et al. [19] address the problem of information leakage on the address bus wherein the attacker would be snooping the address values to gain information about the control flow of the program. They provide a hardware obfuscation technique which is based on dynamically randomizing the instruction addresses. This is achieved through a secure hardware coprocessor which randomizes the addresses of the instruction blocks, and rewrites them into new locations.

Some of the earliest work in this area involves a number

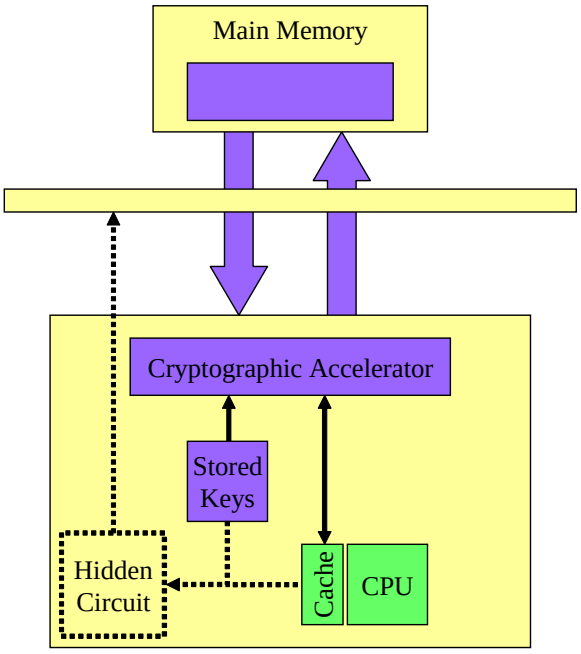

Figure 2. A hidden circuit leaking information

of secure coprocessing solutions. Programs, or parts of the program, can be run in an encrypted form on these devices thus never revealing the code in the untrusted memory and thereby providing a tamper resistant execution environment for that portion of the code. A number of secure coprocessing solutions have been designed and proposed, including systems such as IBM's Citadel [14], Dyad [12], and the Abyss [13].

Smart cards can also be viewed as type of secure coprocessor; a number of studies have analyzed the use of smart cards for secure applications [6, 10]. Sensitive computations and data can be stored in the smart card but they offer no direct I/O to the user. Most smart card applications focus on the secure storage of data although studies have been conducted on using smart cards to secure an operating system [2]. As noted in [1], smart cards can only be used to protect small fragments of code and data.

3. Our Approach

Given this background into these encrypted execution and data systems, one observes that a hidden circuit in the processor can easily leak the keys onto the bus to aid an attacker, as shown in Figure 2.

Our approach includes features at both the architectural and compiler levels. These features can be incorporated into a standard architecture that also runs applications in non-secure mode for the sake of efficiency. Figure 3 shows this dual-use architecture. The regular or non-secure applications execute as usual on a standard CPU, even though the data path from memory to CPU goes through the secure gateway processor (which we describe below). Standard applications are assumed to be constructed using a standard compiler tool chain. The box to the right shows how the same architecture can be applied to peripherals, a matter which we discuss in the following section.

The secure applications are doubly encrypted by the compiler and execute using the dual-processor components of this architecture. One of these processors serves as a gateway whose only function is to perform an initial decryption of the instructions and data before sending the results of this first-level decryption onto the execution processor. The execution processor decrypts this stream a second time to reveal the actual instructions. This part can be performed in a manner similar to the conventional encrypted execution depicted in Figure 1.

Data values that need to be written back to memory are encrypted first by the execution processor and then once again by the gateway using its own keys. Note that the execution processor is physically unable to access the bus; indeed the output of the execution processor is captured by the gateway processor so that any potential leakage can be examined at the gateway.

We assume that all components are individually assumed to be produced at untrusted foundries. Furthermore, we assume that no collusion exists between the foundries that produced the gateway processor and the one that produced the engine processor. We will examine the implications of this assumption in the following section.

We now describe to what extent our approach addresses the foundry threat. A foundry compromise of the execution processor can result in one of three problems:

1. The execution processor can try to write to memory (either to leak information or to disrupt program flow)
2. The execution processor can try to leak information on its pins
3. The execution processor can simply stall or deny service

For the first problem, note that only the gateway is physically connected to memory (the execution processor is physically isolated); the gateway validates all writes to memory. For the second problem, the encryption mechanism at the gateway will detect invalid data. For the third problem, we can use a heartbeat (separately clocked timer) at the gateway to detect denial of service from the execution processor.

Now consider a foundry compromise of the gateway processor. A compromised gateway exposes different system vulnerabilities:

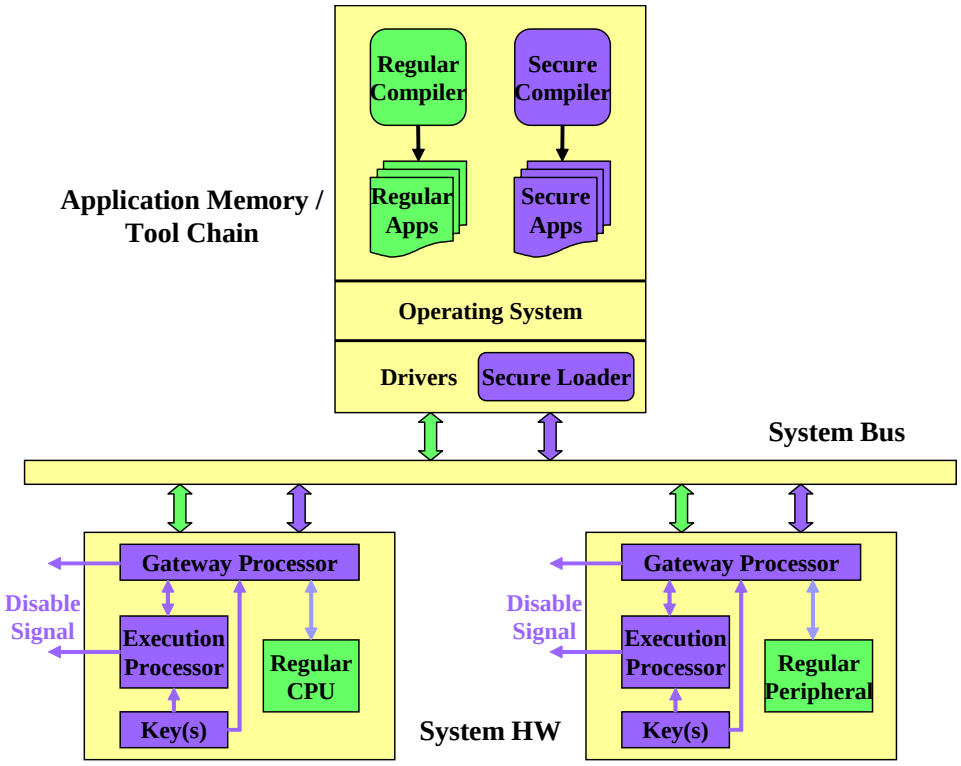

Figure 3. Our approach to obtaining security through untrusted foundries

1. The gateway will be able to leak information on the bus
2. The gateway will be able to write anywhere in memory, including where secure applications reside
3. The gateway will be able to deny service to the execution processor

For the first problem, observe that the gateway can only leak encrypted (by the execution processor) instructions, leaving the instructions secure. For the second problem, note that a write into secure memory can be eventually detected by the execution processor because the gateway is not provided access to the keys for the inner layer of encryption; therefore any overwriting of instructions will be detected as an improper instruction block by the engine. For the third problem, the denial of service can be detected through a heartbeat assigned to the engine processor. Finally, observe that the gateway can attempt a replay attack on the code. This would have to be done by the gateway in a blind fashion since the instructions are encrypted. Such replays can be detected through compiler techniques aimed at detecting control-flow attacks [5, 17].

In sum, the only way for a foundry attack to work is for both chips (foundries) to collude. In the next section, we describe ways in which this risk can be minimized.

4. Discussion

4.1. Validity of Assumptions

Some of the assumptions above are fairly straightforward and constitute standard practice today: (1) system design and implementation given the untrusted chips can be performed in a secure location; (2) there is no collusion between system designers and foundries; (3) encryption is easily performed as part of the compilation process; (4) the two-processor architecture itself imposes no requirement on the software beyond dual-encryption.

However, one might question some of the other assumptions. Of these, perhaps the most troublesome is our assumption that no collusion exists between the foundries that manufactured the two processors. This assumption can be strengthened in several ways, as we now describe. First, we can reduce the possibility of collusion by using two chips made by foundries in different countries. Second, we can use soft processors (with reconfigurable technology) for one or both processor chips so that a foundry has no means of guessing the purpose of the chip. A third technique is to use keys that are programmed post-foundry, using either reconfigurable hardware or by hand using an electromechanical device. When used in combination, these techniques can help minimize the risk of collusion.

4.2. Performance

Although this paper is mainly a concept paper, we can make a few comments related to performance. First, a naive implementation of the two processor architecture may significantly slow performance. An example of a naive implementation would be to implement the cryptographic functions in software. Although straightforward, such an implementation will quickly become the bottleneck and is likely to result in an order of magnitude degradation of performance in applications that are cache-sensitive. A better architectural implementation would involve building custom hardware that pipelines the various units of computation (using pipelined implementations of AES [4, 18], for example) so that the processor is never starved of instructions. At the same time, because the data is encrypted when written to memory, there will always be a substantial performance hit when a cache writeback occurs. This penalty would also be incurred in a standard encrypted execution environment. Proper pipelining and prefetching [8, 9] can help mitigate the effect of our additional encryption.

4.3. System Issues

The system as described above thus far addresses monolithic self-contained applications. Care must be taken to ensure that an operating system can execute in encrypted mode. This is not a trivial task by any means, because of the sometime complex interaction between peripherals, the processor and the operating system. If sensitive data is entered via peripherals, then those chipsets may be just as easily compromised by a foundry. Thus, the architecture needs to be properly extended to peripheral components as well.

5. Implementation Options

5.1. Virtual Machine Implementation

The key principle in our approach combines three ideas: the use of bi-layer encryption, dual mutually-distrusting processors and trusted compilation of secure applications. Note that this principle can be applied not only to hardware (as described above) but to software as well. As an example of the latter, consider a system constructed out of COTS processors running virtual machines. Two virtual machines (whether on the same board, or connected through the Internet) connected in the above architecture can perform the same roles as the two processors, as shown in Figure 4.

There are several advantages to starting with virtual machines: (1) proof-of-concept can be more easily and visibly demonstrated; (2) virtual machines can be instrumented with hooks for red team benchmarking; (3) they can be designed to provide accurate performance estimates; (4) they

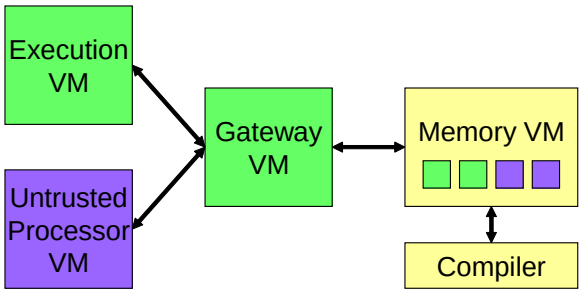

Figure 4. Virtual machine implementation

are easy to modify and replicate, and (5) they are much cheaper than hardware. Each box in Figure 4 represents a virtual machine running in its own process. Two of the boxes represent virtual machines or emulators for the two processors in our secure component. The lower-left box represents the regular (untrusted) processor used for non-secure applications. Finally, on the right side, main memory is shown as containing both secure and non-secure applications. The secure applications are compiled by a trusted compiler that performs the bilayer encryption.

5.2. FPGA Implementation

One interesting and valuable intermediate option between pure hardware and pure software is to use reconfigurable hardware built out of Field Programmable Gate Arrays (FPGAs). FPGAs offer several advantages:

- They can be reconfigured in the field to support multiple instruction sets
- It is harder to hide hidden circuits in them because they are easier to test and their structure is simple
- Their technology curve scales as fast as ASIC technology
- They can be optimized both at the hardware and software levels through intelligent manipulation of design tradeoffs.

We plan on prototyping our hardware system using the Xilinx ML310 FPGA development board (Figure 5). The ML310 contains a XC2VP30 FPGA that has two PPC 405 processors and 13696 reconfigurable CLB slices. The board and FPGA can act as a fully-fledged PC, with 256 MB onboard memory, some solid-state storage, and several standard peripherals.

The flexibility of the Virtex II Pro FPGA allows us to utilize the built-in PPC processors as our standard CPUs, while leveraging soft processors built using reconfigurable logic (such as the Xilinx MicroBlaze [15] processor) as our gateway processors.

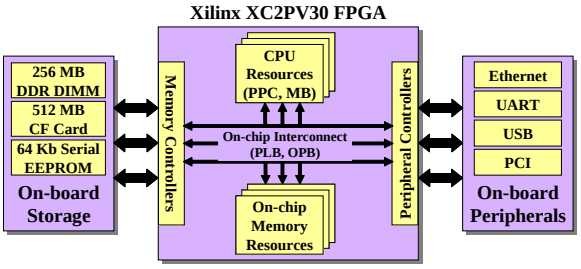

Figure 5. Architectural view of the Xilinx ML310 development platform

6. Summary

We have described the *untrusted foundry problem*, sometimes also known as the *hidden circuit* problem or the *Trojan circuit* problem. We presented a solution to this problem that consists of an architectural component and a software-process (compilation) component. This paper is intended as a concept paper to outline the solution approach. Future work will evaluate the performance of applications to determine the class of applications for which our approach works best.

7. Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grant CCR-0325207, by the Air Force Office of Scientific Research (AFOSR), and also by an NSF graduate research fellowship.

References

[1] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, Nov. 2000.

[2] P. Clark and L. Hoffman. BITS: A smartcard protected operating system. *Communications of the ACM*, 37(11):66–70, Nov. 1994.

[3] International Technology Roadmap for Semiconductors (ITRS). International technology roadmap for semiconductors, 2004 update. available at http://public.itrs.net, Sept. 2005.

[4] K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 207–215, 2003.

[5] D. Kirovski, M. Drinic, and M. Potkonjak. Enabling trusted software integrity. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 108–120, Oct. 2002.

[6] O. Kommerling and M. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20, May 1999.

[7] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, Nov. 2000.

[8] C.-K. Luk and T. Mowry. Cooperative prefetching: Compiler and hardware support fo effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual Internation Symposium on Microarchitecture (MICRO-31)*, Dec. 1998.

[9] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32)*, Nov. 1999.

[10] B. Schneier and A. Shostack. Breaking up is hard to do: modeling security threats for smart cards. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 175–185, May 1999.

[11] W. Shi, H.-H. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 14–24, June 2005.

[12] D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU-CS-91-140R, Department of Computer Scence, Carnegie Mellon University, May 1991.

[13] S. White and L. Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–51, Apr. 1987.

[14] S. White, S. Weingart, W. Arnold, and E. Palmer. Introduction to the Citadel architecture: Security in physically exposed environments. Technical Report RC 16682, IBM Research Division, T.J Waston Research Center, May 1991.

[15] Xilinx, Inc. MicroBlaze processor reference guide. available at http://www.xilinx.com, 2005.

[16] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 351–360, Dec. 2003.

[17] J. Zambreno, A. Choudhary, R. Simha, and B. Narahari. Flexible software protection using HW/SW codesign techniques. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 636–641, Feb. 2004.

[18] J. Zambreno, D. Nguyen, and A. Choudhary. Exploring area/delay tradeoffs in an AES FPGA implementation. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL)*, pages 575–585, 2004.

[19] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, Oct. 2004.