Technical Report No. CE-J09-001, Dept. of Electrical Engineering, Princeton University, Jan. 30, 2009.

INVISIOS: A Lightweight, Minimally Intrusive Secure Execution Environment *

Divya Arora
Intel Corporation
divya.arora@intel.com

Najwa Aaraj
Princeton University
naaraj@princeton.edu

Anand Raghunathan
Purdue University
raghunathan@purdue.edu

Niraj K. Jha
Princeton University
jha@princeton.edu

Abstract

Many information security attacks exploit vulnerabilities in "trusted" and privileged software executing on the system, such as the operating system (OS). On the other hand, most security mechanisms provide no immunity to security-critical user applications if vulnerabilities are present in the underlying OS. While technologies have been proposed that facilitate isolation of security-critical software, they require either significant computational resources and are hence not applicable to many resource-constrained systems, or necessitate extensive re-design of the underlying processors and hardware.

In this work, we propose INVISIOS – a lightweight, minimally intrusive hardware-software architecture to make the execution of security-critical software invisible to the OS, and hence protected from its vulnerabilities. The INVISIOS software architecture encapsulates the security-critical software into a self-contained software module, called the secure core. While the secure core is part of the kernel and is run with kernel-level privileges, its code, data, and execution are transparent to and protected from the rest of the kernel. The INVISIOS hardware architecture consists of simple add-on hardware components that are responsible for bootstrapping the secure core, ensuring that it is exercised by applications in only permitted ways, and enforcing the isolation of its code and data from the rest of the system. We implemented INVISIOS by enhancing the QEMU full-system emulator and Linux to model the proposed software and hardware enhancements, and applied it to protect a commercial cryptograhpic library. Our experiments demonstrate that INVISIOS is capable of facilitating secure execution at very small overheads, making it suitable for resource-constrained embedded systems and systems-on-chip.

*Acknowledgments: This work was supported by NSF under Grant No. CNS-0720110.

1 Introduction

Security has long been the subject of extensive research in the context of computing and communication systems. This has led to developments of various "functional" security measures [1, 2] such as cryptographic algorithms and secure communication protocols. However, security attacks over the years have made it abundantly clear that while functional security measures are necessary, they are not sufficient. Many attacks target weaknesses in a system's implementation and a system's security is only as strong as the weakest link in its implementation.

A recurring theme among many security attacks is that "trusted" code is hijacked at run-time – so even if the original code is not malicious by intent, it can be manipulated by clever attackers, resulting in unwarranted behavior, such as execution of malicious code, leakage of sensitive information, and corruption of program variables. While the subversion of a user application can cause a significant amount of damage, security breaches that exploit vulnerabilities in the OS are especially dangerous. Modern OSs are huge and complex software, spanning tens of millions of lines of code, making it extremely difficult to make them vulnerability-free. Moreover, most commodity OSs are monolithic, implying that the entire OS runs at the highest privilege level. Since there is no privilege separation within the OS, compromise of one part can easily bring down the entire system.

While it may not be feasible to secure all applications from a compromised OS, it may be possible to achieve a *secure execution environment* for selected, security-critical applications (*e.g.*, cryptographic libraries, key management software, integrity verification kernels). Some work has been done towards the above objective, which involves isolation of the critical software from the rest of the system. The isolation may be based on physical separation, wherein the critical software is executed on a physically separate processor and uses a separate memory (*e.g.*, IBM co-processor [3]), or

logical separation, in which both the sensitive and untrusted code run on the same processor, but are isolated, either using an additional layer of software, such as virtualization, or through additional hardware support (*e.g.*, Intel TXT [4] and ARM TrustZone [5]).

In this paper, we present a lightweight, minimally intrusive secure execution environment based on the principle of logical separation. Securing critical software through logical separation is non-trivial since it cannot be achieved by securing the code alone. Even the knowledge of a few intermediate variables, which may be obtained by reading processor registers during context switches, can be used by an attacker to infer the original sensitive data from which they were computed [6]. We focus on providing a secure execution environment for critical software while making minimally intrusive changes to the existing processor architecture. The proposed scheme is especially suitable in the context of SoCs, which are assembled using standard components (including processors), that are provided in the form of intellectual property (IP) cores. In such scenarios, it is difficult for the SoC designer to make intrusive changes to the processor. Our proposal includes add-on hardware units which simply observe the address lines on the main processor chip and base their checking on the observed addresses and certain pre-configured constants. They do not modify the processor pipeline, datapath or other on-chip structures, such as caches and register files. We also do not add any new instructions. The contributions of our work are as follows:

- We propose a set of hardware enhancements to the processor that can facilitate an execution environment that is invisible to, and protected from, the rest of the software that executes on the processor. We refer to the above environment as the *secure domain*, while the non-critical software is run in the *normal domain*.

  1. The hardware enhancements consist of (i) a memory access checker and a bus monitor, which monitor the processor-to-cache bus, and the system memory bus respectively, in order to enforce memory separation between the secure domain and the normal domain, and (ii) a secure bootstrap controller which authenticates and verifies the integrity of the sensitive code, and installs it for execution in the secure domain. We assume the critical software is available as a library, which provides services to other applications.

  2. The software architecture of INVISIOS comprises (i) a user-level stub that provides the same application programming interface (API) as the secure library, encodes the API calls, and makes an appropriate system call, (ii) a kernel-level stub which interfaces between the user stub and the actual library code, and (iii) a secure wrapper that provides single entry/exit points into the library, and makes the secure library self-contained and invisible to the rest of the system.

- We implement a prototype to show the viability of our approach. We incorporate several modifications in the Linux kernel and perform a comprehensive analysis of the penalties imposed by our processor architecture. We analyze the security of our architecture by running the modified Linux kernel on a modified version of the QEMU processor emulator [7].

The rest of this paper is organized as follows. We present a survey of relevant past work in Section 2. We describe the details of the hardware and software components of INVISIOS in Section 3. We present our experimental methodology and results in Section 4. We analyze the security of our architecture in Section 5 and conclude in Section 6.

2 Related Work

Secure execution environments can be achieved by using an additional layer of software underneath the OS to virtualize the hardware platform. Despite significant advances in efficient virtualization, they remain beyond the reach of many resource-constrained embedded systems, which are the primary target of our work. Moreover, the complexity of virtual machine monitors (VMMs) has grown significantly to a point where there is concern about vulnerabilities in the VMM itself [8].

Next, we review past work in the field of hardware-assisted secure program execution.

Researchers have proposed architectural enhancements to validate the control flow of an executing program at run-time and flag deviations from the expected control flow as security exceptions. The enhancements can be in the form of modifications to the processor pipeline [20], dedicated add-on hardware checkers [9], and additions to the processor instruction set [10] that augment the existing instructions to perform additional checking.

Dynamic information flow tracking [11] is a hardware-supported information flow tracker, which tracks the flow of spurious information and prevents it from violating a specified security policy. Secure return address stack [12] is another hardware technique which prevents the disruption of normal control flow affected by a specific type of attack, namely, stack overflow. Secure program execution framework (SPEF) [13] is a combination of architectural and compilation techniques which ensure software integrity and prevent any code, which has not been explicitly installed, from being executed on the system.

The above techniques are primarily designed to protect a benign application from its own vulnerabilities. However, they cannot protect this application, if the vulnerabilities are present in the OS code which lead to compromising the OS.

In light of the above limitations and the fact that modern OSs abound in security vulnerabilities, architectures that offer stronger protection are required for security-critical software, such as cryptographic libraries, digital rights manage-

ment software, *etc.* Some architectures base their security on physical separation, isolating the critical software from the rest of the system and thus protecting it from the weaknesses of the latter. For example, smartcards are often used to execute small computations [14], which involve the use of sensitive code or data. At the higher end of the spectrum are the high-performance secure co-processors which are capable of executing general purpose code and include accelerators for cryptographic operations [3, 15].

It is also possible to create a secure execution environment for critical software based on *hardware- and virtualization-supported logical separation*. In this case, the critical software is executed on the same processor as the OS and other untrusted applications, but the support of hardware or virtual machine monitors is used to maintain a separation between the two. Enhanced processor architectures, such as XOM [16] and AEGIS [17] revamp the processor design to protect applications in the presence of untrusted memory and OS. The attack model assumes that the contents of the memory can be easily read or modified by an adversary and that a compromise of the OS can lead to breakdown of the separation between different processes.

Architectures, such as overshadow [18], aim at protecting the privacy and integrity of application data even when the OS is compromised. They use the concept of multi-shadowing, which gives a different view of the actual machine addresses for different processes. Different views are achieved through the encryption and decryption of memory pages. Secret-protected (SP) architecture [19] and ARM Trustzone [5] aim for a more modest target of a single secure compartment in the processor with fewer changes to the architecture. INVISIOS has a similar objective to SP and Trustzone in the sense that it also creates a single secure compartment with a small amount of hardware support and few modifications to the kernel. However, unlike these approaches, a primary objective of INVISIOS is to make minimally intrusive changes to the processor architecture. Therefore, unlike the above techniques, it does not introduce any new instructions in the processor instruction set architecture (ISA) or modify the tags of on-chip caches and translation lookaside buffers (TLBs). The secure compartment in INVISIOS protects code from unwarranted modification and protects the program data from both unwarranted disclosure and modification. The privacy of program code (which can be achieved through encryption) is not an objective of INVISIOS.

3 INVISIOS Architecture

As mentioned previously, INVISIOS segregates the execution on the processor into two domains: secure and normal. These domains are different from the privilege levels of a program, namely, user and kernel. The software running in the secure domain, referred to as `secure_core` is incorporated in the OS kernel and executes with kernel-level privileges, but is protected from the rest of the kernel by the INVISIOS hardware. In short, the kernel code is a superset of the code executing in the secure domain.

3.1 Threat Model

We assume that the critical code that needs to be secured is available as a software library, `secure_lib`, and only this library needs to be protected by INVISIOS. The applications that use `secure_lib` reside in the non-secure domain and can invoke the library functions through a well-defined API. We further assume that the code of `secure_lib` is trusted, benign, and small enough to be thoroughly checked and made free of security vulnerabilities.

`secure_lib` is encapsulated into `secure_core` which is executed in the secure domain. All user applications and the kernel except `secure_core` are untrusted and execute in the normal domain. The objective of INVISIOS is to ensure that the code and data of `secure_core` are safeguarded from all entities executing in the normal domain. For example, it prevents the malicious overwrite of the `secure_core` code by, say, a compromised OS. In addition, the data of `secure_core`, including globals, heap and stack variables, and temporary values stored in the processor registers during execution, cannot be read or modified by any instructions that do not belong to `secure_core`.

INVISIOS does not protect `secure_lib` against physical attacks that are launched after the library has been loaded. The bootstrap unit (described later) verifies that `secure_core` has not been corrupted before loading. The remaining hardware protects against run-time software-based attacks.

Note that INVISIOS does not limit what `secure_core` can do. Therefore, if `secure_lib` itself is malicious or vulnerable, then INVISIOS cannot preserve the boundary between the secure and normal domains. The remaining sections describe the modifications in the kernel and enhancements to the processor architecture that are required to achieve the above objectives.

3.2 INVISIOS Architecture Overview

The goal of INVISIOS is to provide a secure, isolated execution environment for security-sensitive software while making minimally intrusive changes to the processor. Therefore, no new instructions, cache tags, TLB tags, or modifications to the datapath are introduced. The INVISIOS hardware modules base their checking on the addresses of instruction and data accessed by the processor. The software architecture of INVISIOS also makes relatively few changes to the core OS and is transparent to applications that do not use `secure_lib`.

The core idea behind the design of INVISIOS is to create a self-contained software module, namely, `secure_core` (which encapsulates the code and data to be protected), load it at a fixed physical address in memory [non-accessible by

direct memory access (DMA)] and use hardware support to control access to the memory range reserved for the core. Essentially, the hardware acts as a gatekeeper, opening and closing access to the secure memory range based on certain conditions and events.

Figure 1 depicts the high-level view of the INVISIOS hardware and software architecture. On the software side, the kernel, `secure_core`, and the application are loaded into memory. `secure_core` is part of the kernel and is loaded at a known physical memory address. On the hardware side, the figure shows the main processor, along with on-chip instruction and data caches, and the memory management unit (MMU). The processor and other system peripherals can access the off-chip main memory via the system memory bus. The INVISIOS hardware architecture is shown within the dotted lines.

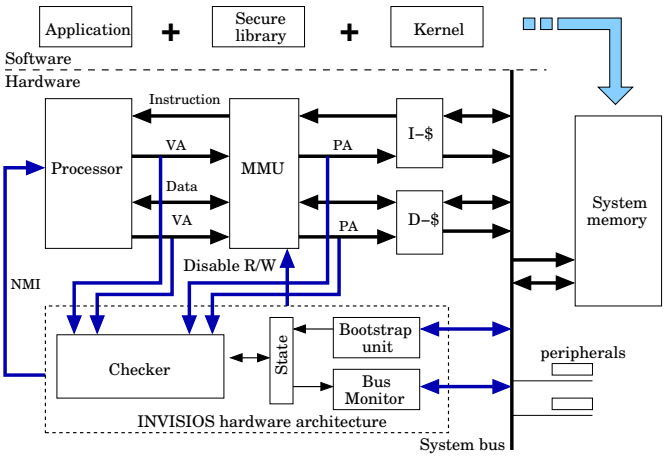


Figure 1. INVISIOS hardware and software architecture

The virtual and physical address lines for both instructions and data are pulled out and fed to the INVISIOS hardware which uses them to enforce the requisite access control. Any violation detected by the INVISIOS hardware resets its state, disables memory access for the current access and results in a non-maskable interrupt (NMI) to the processor. A benign OS can respond to this interrupt by terminating the offending application. In case the OS itself has been corrupted, the result is indeterminate except that no software (not even the OS) can access the secure memory range since access to it is closed by the hardware. Any legitimate software wishing to use `secure_lib` after a violation detection will need to re-initialize the INVISIOS hardware and `secure_core`. The INVISIOS hardware is described in Section 3.4.

Note that Figure 1 shows a processor with separate L1 instruction and data caches. L2 cache has been omitted from the figure for ease of illustration and the INVISIOS architecture is independent of it. Also, the figure shows physically addressed caches, which are addressed using physical addresses after the address translation is completed by the MMU, resident between the processor and the caches. An alternative translation scheme, using virtually-indexed, physically-tagged caches, could use the virtual address to index into the cache and the translated physical address to verify the tag. Even in this scheme, a complete physical address is available before the cache access is considered valid. Therefore, the above cache access scheme is compatible with INVISIOS. INVISIOS is not compatible with virtually addressed caches which are accessed using virtual addresses and MMU translation is performed only in case of a cache miss. However, this is not a major limitation since virtually addressed caches are not commonly used as they create address aliasing problems (multiple virtual addresses mapping to same physical address).

In this section, we describe how a given software library, `secure_lib`, is incorporated into the INVISIOS software architecture. We assume that applications can avail of the services of `secure_lib` through a well-defined API exported by the library. In INVISIOS, the application runs in user space, whereas `secure_lib` runs in kernel space. Shared memory is used to pass application arguments to library functions and to return results from the latter. Usually, the sensitive code which needs to be secured is provided in the form of a user-space library. For example, an application may be partitioned, manually or using data-flow analysis tools, to segregate security-sensitive modules. `secure_lib` needs to be partitioned into two modules – one which runs in the normal domain (in user or kernel space), and is referred to as `normal_secure_lib`, and the other which executes in the INVISIOS secure domain (which resides in the kernel) and is called `invisios_secure_lib`. The parts of the original library, which go into `invisios_secure_lib`, may need to be explicitly ported into the kernel. Note that `secure_core` is a superset of `invisios_secure_lib`.

As mentioned before, a fixed physical memory address range is reserved for `secure_core`, which is protected from user applications and the rest of the kernel by the INVISIOS hardware. `secure_core` comprises `invisios_secure_lib` along with `secure_wrapper` (described below). Every entry into and exit from `secure_core` represent a transition between the two execution domains and must be monitored and validated by the hardware.

Figure 2(a) shows the case when an application is linked with `secure_lib` to produce a regular binary. Calls from the application to the library are regular function calls and do not require OS intervention. Control is transferred to the OS from the application in three cases, as depicted by the block arrows: (a) when the application makes system calls, (b) when an exception is generated by the processor (*e.g*, divide by zero error), and (c) when a hardware interrupt (*e.g*, timer interrupt) is delivered to the processor. A

corrupted OS can take advantage of any of these control transfers to read/corrupt security-sensitive data belonging to secure_lib.

3.3 INVISIOS Software Architecture

Since the OS is untrusted, any control flow or data assignment based on the results of a system call are also unreliable. Therefore, invisios_secure_lib consists only of those parts of secure_lib that do not make any system call. Any kernel subroutine invoked by invisios_secure_lib needs to be replicated in the latter. Cases (b) and (c) mentioned above, in which control is transferred to an untrusted OS, are handled by various components of the INVISIOS software architecture.

Figure 2(b) shows the three main components of the INVISIOS software:

- user_stub: This is a small piece of code which exposes the same API as secure_lib and is linked with the application to produce an executable for INVISIOS. It transforms the library API calls into a system call with appropriate parameters. In the simplest case, the list of parameters comprises the name of the API call and the arguments passed to it by the application. Therefore, invocation of library calls in the binary results in system calls, which transfer control to kernel_stub.
- kernel_stub: This consists of functions that copy arguments from the user space into kernel space, transfer control to secure_wrapper and copy the results returned by the latter back into user space before returning to user_stub.
- secure_wrapper: This is the code which encapsulates invisios_secure_lib and is the main software component responsible for making execution invisible to the OS. secure_wrapper has a single point for entry and a known, finite number of points of exit. Functions in invisios_secure_lib can only be called by secure_wrapper.

  secure_wrapper further consists of three subcomponents: secure_entry, secure_exit, and secure_intr_handler. The secure_entry code saves the caller context (namely, the context of kernel_stub), and changes the address of all the processor interrupt and exception handlers to point to the start of secure_intr_handler. Consequently, any interrupts and exceptions received by the OS during the execution of secure_core are redirected to secure_intr_handler. Finally, the secure_entry code checks if this is a new call into secure_core or whether control has been returned to it after a prior interrupt. In the latter case, it restores the context of secure_core (which was stored by secure_intr_handler when secure_core was last interrupted) and continues execution from the stored return address. In case of a new call, the secure_entry code switches the processor stack pointer to the beginning of the secure stack region (reserved for secure_core), and invokes the appropriate function in invisios_secure_lib.

  secure_intr_handler is invoked when an interrupt or exception occurs while secure_core is executing. Its primary functions are to save the context of secure_core, zeroize registers used for holding temporary values, create a fake interrupt context, restore the original interrupt handlers and jump to the original handler corresponding to the current interrupt of exception. Essentially, secure_intr_handler just performs some cleanup functions before letting the original handler handle it. The fake interrupt context is created in such a way that the original interrupt handler believes that the interrupt came when the program was executing in kernel_stub and returns control to the latter after completing its function. kernel_stub then jumps to the single point of entry into secure_wrapper, and execution resumes.

  The secure_exit code cleans up the state set up by secure_entry. It resets the saved secure_core context, restores the original interrupt handlers and returns to kernel_stub, which, in turn, returns to user_stub. invisios_secure_lib does not invoke any outside kernel functions, and all interrupts and exceptions occurring during the execution of secure_core are caught by secure_intr_handler before passing control to the original handlers. Thus, the OS is oblivious of secure_core and only knows about kernel_stub to which it returns control after the original handler has completed its job.

In all the components described above, interrupts are disabled and re-enabled as necessary so that an interrupt during their operation does not affect the correctness or security of the system. secure_wrapper needs to be part of the kernel since it changes interrupt handlers, which involves the use of privileged instructions. The three-layered software architecture facilitates a smooth transition from user_stub running with minimal privileges to secure_core which runs with the highest privileges as part of the kernel, but is segregated in memory from the rest of the kernel and is protected from it with the help of the INVISIOS hardware.

INVISIOS trades off flexibility for simplicity. The fact that secure_core lies within a single contiguous memory range simplifies the hardware checker greatly. However, it also imposes several restrictions on the code contained in secure_core. Since it has to be self-contained and be limited to a certain memory range, secure_core cannot call an external dynamic memory allocator. Therefore, we implement a simple memory allocator in secure_core which al-

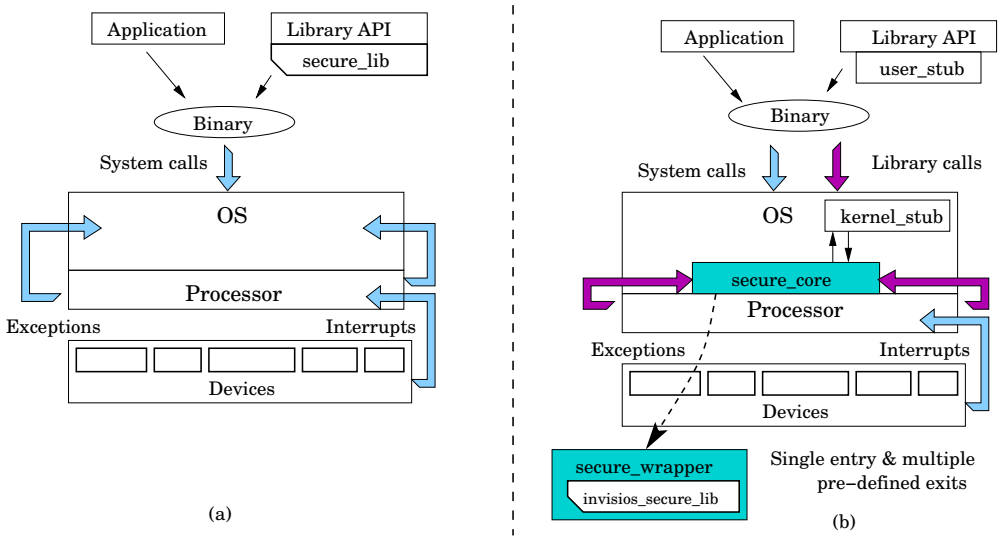


Figure 2. (a) Regular software architecture for applications using secure library, and (b) INVISIOS software architecture

Table 1. State registers in INVISIOS hardware

Register	Description
K_{pub}	Public key of the certifying authority permitted to sign software for INVISIOS
$R_{command}$	Used to command the bootstrap unit to install a new library
$R_{start\_address}$	Start address in memory of the new library to be installed by the bootstrap unit
R_{init_state}	Set after the bootstrap unit has authenticated the secure library and initialized other registers
R_{entry_point}	Address of the single point of entry into `secure_core`
$R_{exit\_interrupt}$	Address of the instruction at which `secure_core` exits on an interrupt
R_{exit_final}	Address of the instruction at which `secure_core` exits after completing its operation
R_{secure_state}	Set if the processor is executing code in `secure_core`. Access to the secure memory region by the processor is permitted at this time
$R_{high\_address}$	Upper bound of the address range in which `secure_core` is contained
$R_{low\_address}$	Lower bound of the address range in which `secure_core` is contained

locates memory out of a memory pool which is reserved for it and returns freed memory to the pool. `secure_core` interfaces with the application through shared memory which is assumed to lie in the normal domain and can be of unrestricted size. Also, some memory pages are statically reserved for the program stack of the core which is managed by `secure_wrapper`. We recognize that this is a limitation since it is hard to bound the stack size of an application. Work is ongoing to permit contiguous physical memory pages to be dynamically added and removed from the memory range reserved for the core, in a secure fashion.

3.4 INVISIOS Hardware Architecture

The INVISIOS hardware comprises functional units that perform various kinds of checking and maintain its state information. The state is stored in programmable registers which can be set only by other INVISIOS functional units. In addition, the hardware comes pre-programmed with the public part of a public-key pair, K_{pub}. The key is stored in non-volatile memory and is used for initial authentication of `secure_core`. Only a `secure_core` that is signed by the private part of the key pair, K_{pvt}, can be installed on the INVISIOS system. K_{pvt} is known to a certifying authority, to which software vendors desiring to create an INVISIOS core can send their code for signing.

A description of the state registers is given in Table 1. At any stage, any of the active functional units of the INVISIOS hardware can report a violation. This results in an NMI to the processor and resets registers R_{init_state} and R_{secure_state}. The main functional units of INVISIOS hardware are illustrated in Figure 3 and described in the following subsections.

3.4.1 Bootstrap unit

The bootstrap unit is responsible for installing a new `secure_core` (and, therefore, a new `secure_lib`) in the protected memory. It authenticates `secure_core`, verifies its integrity, and initializes the hardware registers with parameters provided in the core. The OS copies the con-

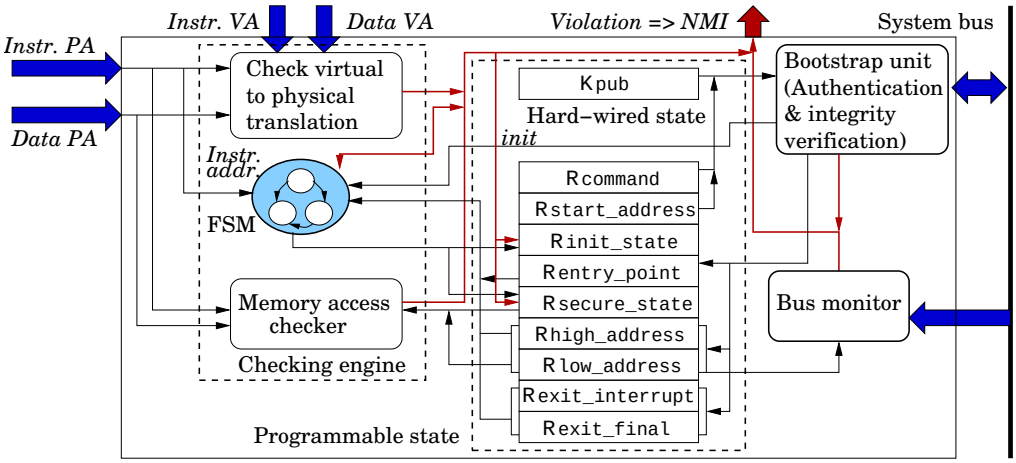


Figure 3. INVISIOS hardware architecture

tents of `secure_core` into physical memory at address `core_start_addr`, writes the same address to register $R_{start\_address}$, and directs the bootstrap unit to begin installation via $R_{command}$. The core is not usable until it has been explicitly installed by the bootstrap unit.

`secure_core` is created as described in the previous section. Its header contains the size of the image and the hash of the image signed using K_{pvt}. The bootstrap unit computes the hash of the core, decrypts the signed hash using K_{pub} and compares the two to verify the authenticity and integrity of loaded `secure_core`. If the above operations go through without an error, it asserts the *init* signal to the finite-state machine (FSM, described below), and initializes registers R_{entry_point}, $R_{exit\_interrupt}$, R_{exit_final}, $R_{high\_address}$ and $R_{low\_address}$, with values specified in the core image.

The value in R_{entry_point} denotes the address corresponding to the single point of entry into `secure_core`. Registers R_{exit_final} and $R_{exit\_interrupt}$ store the two exit points from `secure_core`: the former stores the address of the last instruction in the core after which it returns to `kernel_stub`, having completed its operation; the latter stores the address of the instruction in `secure_intr_handler` which transfers control to the original handler, when an interrupt (or exception) occurs during the operation of `secure_core`. $R_{high\_address}$ and $R_{low\_address}$ designate the memory range reserved for the core which is protected by the checker. After initialization of the registers, the bus monitor is activated.

3.4.2 Checking engine

The checking engine is the primary hardware module which controls access to `secure_core`. There are no separate commands to instruct the checking engine to start or stop checking for access violations. The checking engine is "always on" after the initialization phase. It has three major components:

- **FSM to enforce the `secure_core` interface**: The FSM is a small control unit which ensures that the entry into and exit from `secure_core` can only happen at permitted addresses. For example, it checks that the execution of `secure_core` can begin only at a fixed address programmed into R_{entry_point}. If a program tries to jump into `secure_core` at any other address, the FSM reports a violation. Once the program has entered `secure_core`, the FSM sets R_{secure_state} indicating that the execution is in the secure domain and access to all memory is permitted at this time. Similarly, the FSM ensures that the exit from `secure_core` can only be at one of the two exit points defined above.

  The FSM has three states: (i) `START`, which is the initial state of the FSM where secure core is not yet usable and needs to be verified by bootstrap unit before use, (ii) `INIT`, which is the state to which the FSM goes after initialization by the bootstrap unit and after exit from `secure_core`, and (iii) `SECURE`, which is the state of the FSM when the secure core is executing. The proposed scheme has a very simple FSM that imposes minimal checks. However, it can be easily augmented with more complex checking mechanisms, such as those proposed in [9,20], to incorporate complete control flow checking of `secure_core`.
- **Virtual to physical address translation checker**: In processors, such as XOM and SP, secure versions of load/store instructions accompanied by cache and TLB tags are used to ensure that a program is writing to or reading from memory in its own secure compartment. If the process wants to share data with another process, it writes to a special compartment which is untagged and unchecked.

  In INVISIOS, there are no special instructions to read from or write to the protected memory range. Also, `secure_core` is permitted to access both regular and

protected memory. Most of its writes are to protected memory except when it is writing back the results of computation to the shared memory so that they can be accessed by the calling application.

Consider the scenario when a compromised OS alters the virtual to physical address translation, thereby tricking `secure_core` into writing to unprotected physical memory under the false assumption of security. In order to prevent such an attack, a checker is required, which verifies the address translation.

In the user mode, a virtual address in the executing program can map to any physical address and these mappings are stored as entries in the page table. Therefore, verifying the address translation of code running in the user space would require storing the entire page table in hardware. However, in INVISIOS, `secure_core` runs in kernel mode. In kernel mode, the physical address is at a fixed offset from the virtual address. Therefore, the hardware can verify the translation by a simple addition rather than comparing it against pre-stored page table entries.

- **Memory access checker**: This functional unit prevents any code from outside the secure core from accessing any word lying in the protected memory range. It validates each load and store instruction executing on the processor and allows or disallows access based on the state of the INVISIOS hardware. If R_{secure_state} is set, the checker permits the processor to access any memory region. However, if R_{secure_state} is reset, and the processor attempts to access memory lying between the address in $R_{low\_address}$ and $R_{high\_address}$, the checker reports a violation and sends an NMI to the processor.

3.4.3 Bus monitor

The bus monitor is a watchdog module which prevents potentially malicious bus peripherals from reading/writing to the protected memory range. The checking engine provides similar protection, but its protection extends only to code executing on the processor. The bus monitor extends this protection to peripherals which can access memory using DMA. It monitors the traffic on the system memory bus and flags accesses to the protected memory by any peripheral as a security violation. The offending bus transaction is aborted as a result of this, and the INVISIOS registers and memory state are reset. The bus monitor is activated by the bootstrap unit and is also "always on," like the checking engine.

4 Experimental Methodology and Results

In this section, we present details of our experimental framework for implementing and testing INVISIOS.

4.1 INVISIOS Software Architecture

We implemented the INVISIOS software architecture in Linux kernel version 2.6 for the x86 architecture. In our implementation, `kernel_stub` is a Linux device driver which interfaces with applications using regular `open`, `close`, `read`, `write` and `mmap` system calls. `kernel_stub` preprocesses arguments and jumps to the single point of entry in `secure_wrapper`. `secure_wrapper` performs the functions describe in Section 3.3 and calls the appropriate function in `invisios_secure_lib`.

`secure_lib` is a commercial cryptographic library *CLIB* [1] which was ported to the kernel. It supports a wide range of cryptographic algorithms including symmetric key algorithms (DES, 3DES, AES, RC4), one-way hash functions (MD5, SHA-1), public key algorithms (RSA, Diffie-Hellman, Elliptic curve), and digital signature algorithms (DSA, RSA, Elliptic curve).

The original library, *CLIB*, is structured in such a way that its core runs in a different address space from the calling application. The execution follows a client-server model, in which the core of the library runs as a server, waiting for application requests. The *CLIB* client consists of the application linked with a stub library which exposes the *CLIB* API. Shared memory is used to pass arguments and results between the client and the server, and semaphores are used for synchronization between the two. In INVISIOS, the server code is migrated to the kernel and forms `invisios_secure_lib`. The calls in the client code which generate computation requests are converted to calls to the INVISIOS device driver. The device driver acts as a communication link between the client running in user space and `invisios_secure_lib` running in kernel space in the INVISIOS secure domain.

As mentioned before, memory space is statically allocated for the data and stack of `secure_core`. Physical memory pages are reserved for the core and pinned along with the rest of the kernel. We use a simple dynamic memory allocator to dynamically allocate and free memory from a fixed pool. Linker scripts were used to generate a kernel image which has a contiguous memory range reserved for `secure_core`. Kernel symbol table information was extracted to determine the addresses of INVISIOS entry and exit points, which were used as configuration parameters for the INVISIOS hardware.

In order to achieve speedy switching between interrupt handlers in `secure_wrapper`, two copies of the interrupt descriptor table (IDT) were maintained. The address of the IDT is stored in a processor register called IDTR (interrupt descriptor table register), whose contents are modified by `secure_wrapper` to change interrupt handlers. In case of interrupts, the x86 processor pushes the contents of eFlags, CS (code segment) and EIP (instruction pointer) registers

[1] Name changed to protect vendor.

onto the stack before jumping to the appropriate interrupt handler. In case of exceptions, such as page fault, stack exception, *etc.*, an additional error code is pushed onto the stack by the processor. `secure_intr_handler` implements different functionalities for different interrupts and exceptions. After zeroizing processor registers and performing other cleanup operations, it creates a fake interrupt/exception context as expected by the original handler before jumping to the latter.

The modified kernel was run on a Pentium 3.4GHz laptop. Test applications were selected from *CLIB's* regression test suite and run on top of the modified kernel. Both the kernel and test applications were compiled with "-o2" optimization flag. The test programs exercise various parts of *CLIB*, including symmetric key encryption, cryptographic hash computation, and public key operations. Figures 4(a), (b) pertain to hash computation using SHA-1 and MD5 hash algorithms for varying data sizes, Figure 4(c) presents data for RSA key generation with varying sizes for the modulus, and Figures 4(d–f) illustrate the performance of AES CBC (cipher-block chaining), and 3DES CBC and ECB (electronic code-book) encryption and decryption.

The time measurements were made using the Pentium time-stamp counter, which is a 64-bit register, incremented on every clock cycle. It maintains an accurate count of the processor cycles and can be accessed using the `rdtsc` instruction. The average performance penalty in the tested benchmarks was 1%.

4.2 INVISIOS Hardware Architecture

The INVISIOS hardware was modeled on a full-system emulator QEMU [7]. QEMU is a multi-processor emulator (*i.e.*, it supports multiple host and target processors) which achieves good emulation speed using dynamic binary translation. In our study, we incorporated several modifications in the x86 emulation engine in order to include the INVISIOS hardware, as follows:

1. Addition of registers R_{entry_point}, $R_{start\_address}$, R_{entry_point}, $R_{high\_address}$, and $R_{low\_address}$.
2. Addition of a hardware checker, which implements virtual to physical address translation checking and memory access checking against the values stored in $R_{low\_address}$ and $R_{high\_address}$.

We tested the functional correctness of INVISIOS by running the modified Linux kernel (described above) on QEMU. The INVISIOS hardware implemented within QEMU did not report false positives for any benchmark. We also created malicious modifications in the kernel to access INVISIOS memory when the processor was not executing in the secure domain. Malicious modifications of the kernel include:

1. Modification of virtual to physical memory mappings in the mm/ Linux folder.
2. Accessing memory addresses lying between the addresses in registers $R_{low\_address}$ and $R_{high\_address}$.
3. Modification of FSM registers.

All malicious accesses to INVISIOS memory space were caught and flagged by the hardware checker added to QEMU.

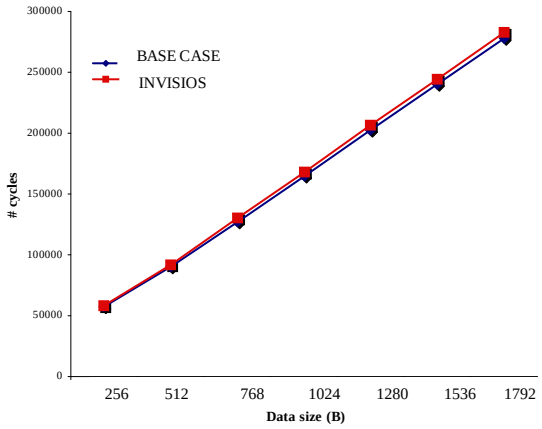
Hardware modeling was done for the purpose of functional testing only. Performance estimates for INVISIOS hardware could not be obtained due to lack of availability of cycle-accurate simulators which can execute an entire OS. However, one of the main considerations in the design of INVISIOS was simplicity. Since the INVISIOS hardware is composed of simple checkers, they can perform the checking in parallel with program execution without causing noticeable performance penalties. We do not expect a performance impact in addition to the impact of the INVISIOS software architecture presented in Section 3.3.

5 Security Analysis

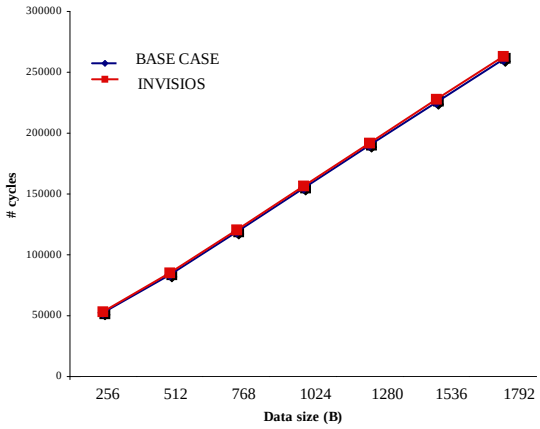
In this section, we analyze the security of our architecture and show how it blocks attacks against the secure library that is being protected. In our architecture, the code of the untrusted application is inherently protected from modification. However, an attack can target vulnerabilities in the OS to access the secure core memory or to cause the INVISIOS to leak confidential data by storing it in unprotected physical memory . Other attacks could use DMA accesses, bus peripherals, and page faults to read or write to the protected memory range.

We first discuss the various forms of attacks aimed at accessing the secure core memory. These attacks work by (i) maliciously modifying the OS virtual-to-physical address translation mechanism (Attack 1), (ii) accessing protected memory from code lying outside the secure core (Attack 2), (iii) maliciously modifying bus peripherals to access secure memory through DMA (Attack 3), (iv) exploiting page faults to page out memory pages belonging to the secure core (Attack 4), or (v) exploiting vulnerabilities in the secure library to leak information (Attack 5).

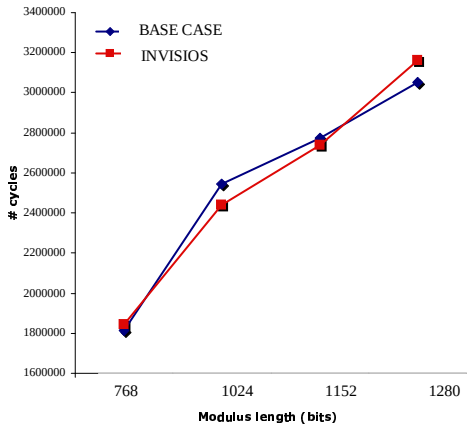
Attack 1 is effectively blocked by the virtual-to-physical address translation checker, which, instead of storing the entire page table in hardware and updating it upon page faults, verifies the translation by a single offset addition. Attack 2 is also blocked by the memory access checker, which validates each load and store instruction executing on the processor and disallows access to the secure memory core (*i.e.*, any memory address between the value of $R_{low\_address}$ and that of $R_{high\_address}$) from outside the core if R_{secure_state} is set. A more sophisticated type of attack exploits DMA to access secure memory. This attack (Attack 3) is circumvented in two ways: (i) through the bus monitor, which flags access by peripherals to protected memory as a violation, and (ii) by specifying that the memory region between $R_{low\_address}$ and $R_{high\_address}$ is not accessible through DMA. Attack 4 is circumvented by pinning the secure physical memory along



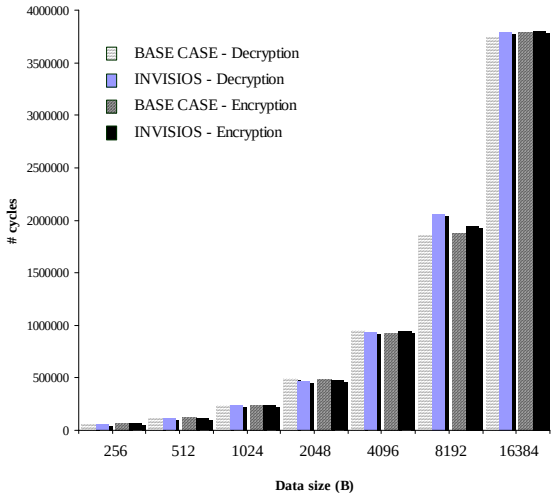
(a) SHA-1 hash



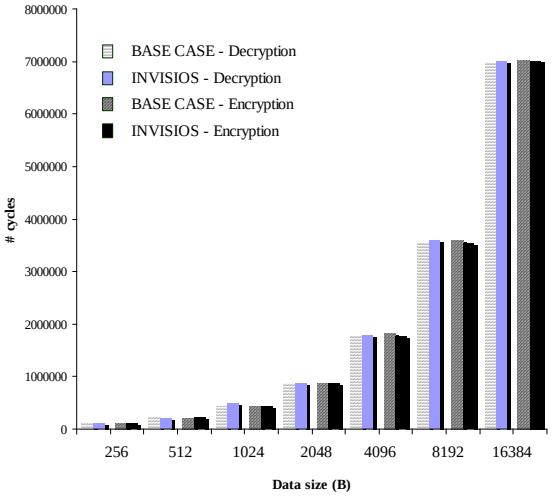
(b) MD5 hash



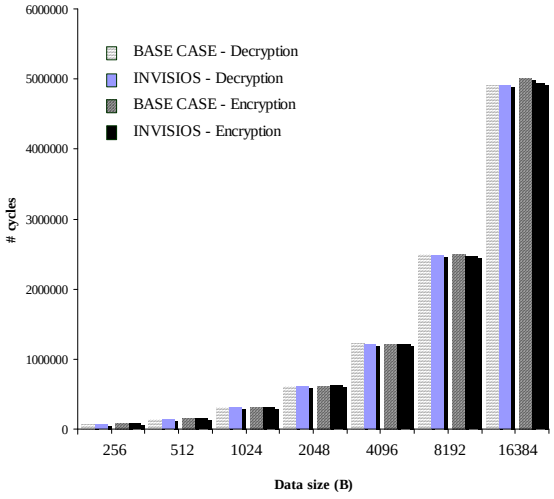
(c) RSA key generation



(d) AES CBC encryption and decryption



(e) 3DES CBC encryption and decryption



(f) 3DES ECB encryption and decryption

Figure 4. Performance results for INVISIOS

with the kernel, thus, the memory used by the secure core cannot be paged out. If an OS handler tries to swap out a `secure_core` page and replace it by another, the memory access handler is able to detect it as an illegal access.

INVISIOS does not aim to protect against Attack 5, since it assumes that the code in `secure_core` is trusted and verified to not leak sensitive information. INVISIOS allows `secure_core` to read and write outside the secure memory, which is necessary for the secure core to communicate with the calling application. However, this is a common limitation of all secure execution environments, including those based on virtualization or extensive processor enhancements.

6 Conclusions

In this paper, we presented INVISIOS, a hardware-software architecture for providing a secure execution environment for security-critical software. INVISIOS makes the execution of critical software invisible to the OS and thus makes the former immune to vulnerabilities of the latter. The INVISIOS software architecture requires very few changes in a commodity OS. The hardware enhancements entailed by INVISIOS are non-intrusive (requiring no new instructions, cache/TLB tags or changes in the datapath) and result in a very simple checker design. We have implemented INVISIOS by running a modified Linux kernel on an x86 emulator modified to include INVISIOS hardware. This, along with our performance measurements on an actual x86 platform, indicates that the proposed architecture provides an effective way to achieve a secure execution environment on a regular processor, at reasonable overheads.

References

[1] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.

[2] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.

[3] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks*, vol. 31, no. 9, pp. 831–860, Apr. 1999.

[4] *LaGrande technology preliminary architecture specification*. Intel Publication no. D52212, May 2006.

[5] *ARM TrustZone Technology Overview*. `http://www.arm.com/products/CPUs/arch-trustzone.html`.

[6] N. R. Potlapally, A. Raghunathan, S. Ravi, N. K. Jha, and R. B. Lee, "Satisfiability-based framework for enabling side-channel attacks on cryptographic software," in *Proc. Design, Automation and Test in Europe Conf.*, Mar. 2006, pp. 18–23.

[7] "QEMU open source processor emulator." `http://fabrice.bellard.free.fr/qemu/`

[8] J. McCune, B. Parno, A. Perrig, M. Reiter, and A. Seshadri, "Minimal TCB code execution," in *Proc. Int. Symp. Security and Privacy*, May 2007, pp. 267–272.

[9] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted runtime monitoring," in *Proc. Design, Automation and Test in Europe Conf.*, Mar. 2005, pp. 178–183.

[10] M. Budiu, Úlfar Erlingsson, and M. Abadi, "Architectural support for software-based protection," in *Proc. Wkshp. Architectural and System Support for Improving Software Dependability*, Oct. 2006, pp. 42–51.

[11] G. E. Suh, J. Lee, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, July 2004, pp. 85–96.

[12] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A processor architecture defense against buffer overflow attacks," in *Proc. Int. Conf. Information Technology: Research and Education*, Aug. 2003, pp. 243–250.

[13] D. Kirovski, M. Drinić, and M. Potkonjak, "Enabling trusted software integrity," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 108–120.

[14] M. J. Atallah and L. Jiangtao, "Enhanced smart-card based license management," in *Proc. Int. Conf. E-Commerce*, June 2003, pp. 111–119.

[15] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. W. Smith, "Building the IBM 4758 secure coprocessor," *IEEE Computer*, vol. 34, pp. 57–66, Oct. 2001.

[16] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 168–177.

[17] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. Int. Conf. Supercomputing*, June 2003, pp. 160–171.

[18] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2008, pp. 2–13.

[19] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proc. Int. Symp. Computer Architecture*, June 2005, pp. 2–13.

[20] T. Zhang, X. Zhuang, S. Pande, and W. Lee, "Anomalous path detection with hardware support," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, Oct. 2005, pp. 43–54.