Architectural Enhancements for Secure Embedded Processing

Divya ARORA[a], Srivaths RAVI[b], Anand RAGHUNATHAN[b] and Niraj K. JHA[a]
[a]*Dept. of Electrical Engineering, Princeton University, Princeton*
[b]*NEC Laboratories America, Princeton*

Abstract. In this paper, we present architectural enhancements for ensuring secure execution of programs on embedded processors. The primary motivation behind this work is that software attacks often originate from unknown vulnerabilities in trusted programs. We propose two techniques to achieve secure program execution. They include (i) hardware-assisted monitoring of a trusted program's control flow to detect deviant control flow, and (ii) hardware-assisted validation of a program's data properties that may be violated in the event of an attack. Experiments show that the proposed architecture can be very effective in preventing a wide range of security threats.

1. Introduction

Various studies [1,2,3] indicate that software attacks today are not only increasing in number, but are also beginning to target a diverse range of electronic systems. Software attacks due to viruses, worms, and trojan horse applications have proliferated not only to personal computers, but also to embedded appliances such as cellphones and PDAs, automotive electronics, and networked sensors. This trend is attested to by the emergence of attacks on mobile phones such as the Skulls, Cabir, and Lasco viruses [4]. Given that embedded systems are ubiquitously deployed in several mission-critical and pervasive applications, it is not surprising that security of an embedded system is becoming a major and immediate concern to manufacturers and users, alike.

In the desktop and enterprise world, conventional software security solutions such as software certificates, software vulnerability patches, ant-virus software updates, *etc.* have achieved only limited success. This observation has led to the development of various architectural mechanisms to augment software security solutions, and many of these technologies are emerging today as commercial products/solutions. For example, processors from Intel and AMD now feature a non-executable bit that can be enabled to make selected regions of a program's address space non-executable [8]. This makes the programs that they execute less vulnerable against buffer overflow attacks. Similarly, chips designed for next-generation cellphones, such as TI's OMAP 2420 [6] and NEC's MP211 [7] systems-on-a-chip (SoCs) feature a wide range of security measures including secure bootstrapping, and protection of the code and data spaces associated with sensitive applications. Another recent technology applicable to embedded SoCs is ARM's TrustZone [5], which attempts to

provide a secure execution environment for a selected set of applications called trusted applications. The basic objective is to provide protection for the code and data spaces of trusted applications against tamper by untrusted applications.

Our work also falls in the domain of architectural support for security. The objective is to provide higher security assurance when a trusted program executes, so that the system can be protected against software attacks that can even originate from a vulnerability in the trusted program. We have developed two techniques to address this objective, which include:

- *Hardware-assisted control flow monitoring:* Many attacks, such as stack-based buffer overflows, execute malicious code by exploiting vulnerabilities in a trusted program. Protecting against such attacks is, therefore, a critical objective. Our solution is based on a simple observation that the execution of malicious code will result in behavior or control flow that is different from the normal program behavior. In other words, if the trusted program can be characterized to have a normal execution behavior (in the absence of an attack), then any deviation from the said behavior can be flagged as an attack. We implement the proposed solution by designing a separate hardware monitor that models and enforces the characterized program behavior by monitoring the program's execution on the processor. The proposed framework shows that a program's function call graph, basic block control flow graph, *etc.* are invariants that can be statically derived, and enforced by the monitor at run-time, with minimum overheads, and in a minimally intrusive fashion.
- *Hardware-assisted validation of program's data properties:* Some attacks do not modify the control flow of a program, but only modify the data associated with a program in the program's stack or heap. Similar to control flow, a program's behavior with respect to data accesses can also be encoded and enforced as security policies during the program's execution. This led to the development of a HW/SW framework that can enforce various security policies. This framework is shown to be effective in preventing various kinds of software attacks, including heap-based and format string attacks.

The rest of this paper is organized as follows. Section 2 discusses related work. Sections 3 and 4 detail the two techniques. Section 5 presents experimental results, and Section 6 concludes the paper.

2. Related Work

Many techniques including code scan and review tools attempt to strengthen software security by eliminating vulnerabilities during the software design phase. Various solutions have been developed to address specific kinds of attacks such as stack-based buffer overflow attacks [9], heap overflow attacks [10], *etc.* Apart from such mechanisms, researchers have proposed a wide range of runtime monitoring techniques [11, 12] to enforce various security policies. However, software-based runtime monitoring techniques

suffer from various drawbacks including performance overheads, limited coverage of security vulnerabilities, *etc.*

More recently, researchers have focused on augmenting processor architectures for secure program execution. Examples of these works include enhanced processor architectures, such as XOM [13] and AEGIS [14], which attempt to provide code integrity and privacy in the presence of untrusted memory. However, these techniques do not safeguard an application from its own vulnerabilities. A more detailed survey of work related to this paper can be found in [15] and [16].

3. Hardware-assisted Control Flow Monitoring

We first present an example attack to motivate the proposed architecture, which we then describe in detail.

3.1. Example Attack

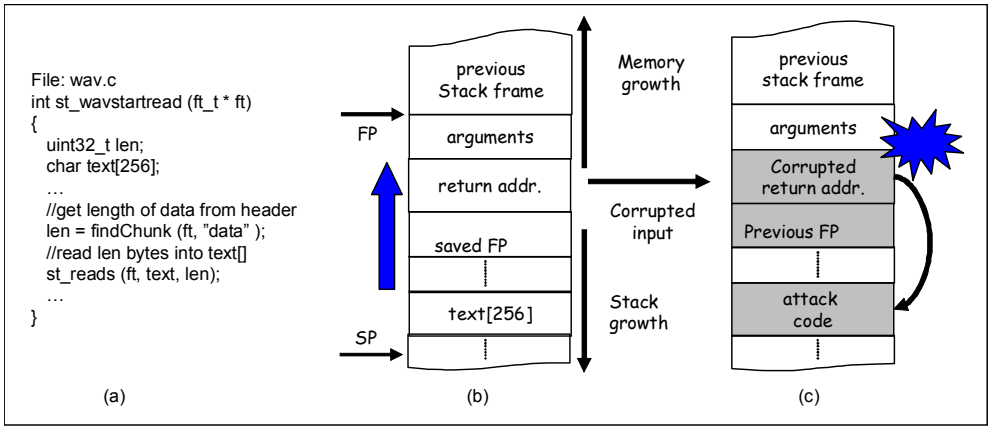

Fig.1. A simple example of a "stack smashing" attack

Fig.1(a) presents a code snippet from SoX (Sound eXchange), a popular audio conversion utility. The function *st_wavstartread()* reads *len* bytes from an input file into a local array *text[]*. Fig. 1(b) shows the stack layout, when the function is called during program execution. An attacker creates an input *wav* file containing a payload of malicious code and *len>256*. This causes a buffer overflow for *text[]*, resulting in corruption of the local variables and function's return address stored on program stack. The input file can be constructed so that the corrupted return address points to the start of malicious code, which is executed when the function *st_wavstartread()* returns.

While the vulnerability in the above example could be easily addressed through input validation, bugs in large, complex programs can be much more subtle and elusive. In addition to software attacks, embedded systems are also susceptible to physical attacks that involve tampering with system properties such as voltage levels and memory contents. Irrespective of their origin, most attacks manifest as a subversion of "normal" program

execution. Therefore, we concentrate our efforts on defining this behavior and monitoring execution to enforce it, using hardware support to make such extensive checking feasible.

3.2. Proposed Architecture

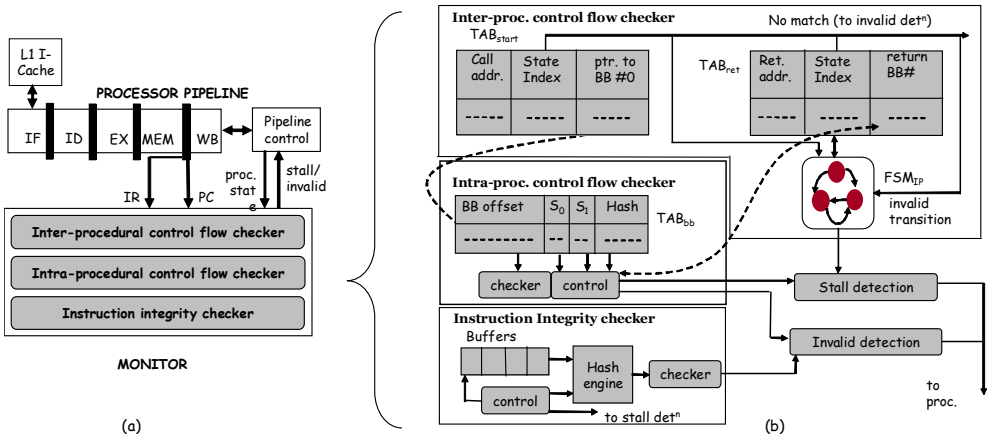

Fig.2. Hardware-assisted monitoring: (a) Block diagram, and (b) Detailed architecture

Fig. 2(a) shows the block diagram of the proposed hardware-assisted monitoring architecture. The figure shows a 5-stage, in-order RISC processor pipeline that has been enhanced with a separate hardware unit or *monitor*. The monitor receives inputs from the program counter (PC), instruction register (IR) and the pipeline status from the pipeline control unit. The monitor's outputs include a *stall* signal and an *invalid* signal. When the monitor detects a violation of permissible behavior, it asserts the *invalid* signal resulting in a non-maskable interrupt to the processor. The *stall* signal is asserted if the monitor is unable to keep pace with the processor. This is handled as a normal processor stall, and the pipeline stages are "frozen" till the stall signal is de-asserted.

The monitor is composed of three sub-blocks, which check program properties at different granularities. The structure is hierarchical and follows from the natural structure of programs – the top-most level works at the application level and verifies that the *inter-procedural control flow* is in accordance with the program's static function call graph, the second level validates the *intra-procedural control flow* by validating each branch/jump instruction within a function and finally, the lowermost level verifies the *integrity of the instruction stream.* This hierarchical structure allows the designer to trade-off checking granularity and coverage for area and/or performance.

Fig. 2(b) details the design of each of these sub-blocks. The call graph of the program is modeled as a finite state machine (FSM), with each function represented as a state and caller-callee relationships denoted by valid FSM transitions. The FSM monitors all call/return instructions and transitions to the next state accordingly. There is a common invalid state to which the FSM transitions if a function attempts to call or return to another function it was not permitted to. Tables TAB_{call} and TAB_{ret} maintain the mapping from program addresses to FSM state identifier and feed it to the FSM. Intra-procedural control

flow is also modeled as an FSM with a state for each basic block in the control flow graph of the function. The information for a basic block b_i is represented as a tuple *(index,offset,S_0,S_1)* where *index* is its state identifier, *offset* is the address offset of b_i from function start, and S_0, S_1 are indices of basic blocks that are its possible successors. This FSM is stored in the table TAB_{bb} and the associated control/checking logic verifies that the branches and jumps made during execution correspond to the information stored in this table. The instruction integrity checker checks the integrity of executing instruction stream by computing a cryptographic hash of each basic block and comparing it against a pre-specified value. For this, the program loads the basic block hashes for a set of (sensitive) functions into the monitor before execution and the monitor verifies that the instructions are not modified during execution. This sub-block consists of a pipelined hashing unit and buffers to store the instructions while the hash of a previous basic block is being computed, so as to not stall the processor too frequently.

4. Hardware-assisted Validation of Program Data Properties

We first use an example attack to motivate the proposed architecture, which we then describe in detail.

4.1. Example Attack

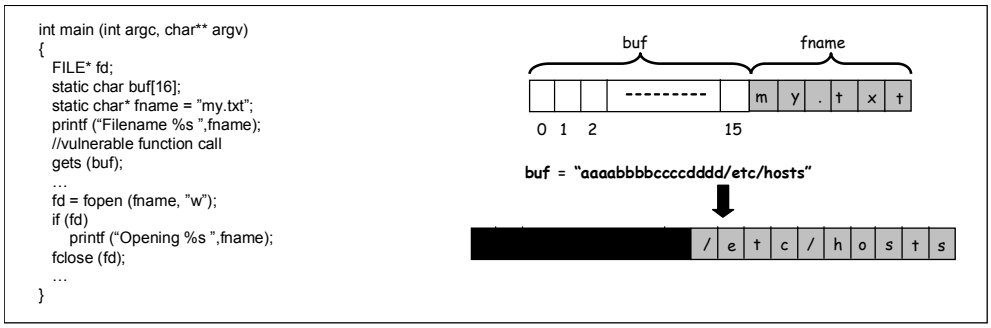

Fig.3. Program data corruption via heap overflow

Fig. 3 shows a typical heap overflow attack. The code contains two static variables, *buf[]* and *fname* on a program's heap. The program uses a vulnerable C function *gets()* to get input from the user and store it in *buf[]*. A malicious user can provide a long input that overflows *buf[]* and overwrites **fname*, tricking the program into writing to a completely different (sensitive) file. There are two points worth noting here: (i) it is possible to corrupt program data without altering its control flow behavior, and (ii) neither the OS nor the processor architecture provide any mechanism to prevent transgressions by one part of an application against another. We address these two limitations in this work in the context of program data protection. Data protection is significantly harder than protecting code, since unlike code (which is mostly known after compilation), data are highly dynamic, and may be allocated, initialized, and changed at run-time.

4.2. Proposed Architecture

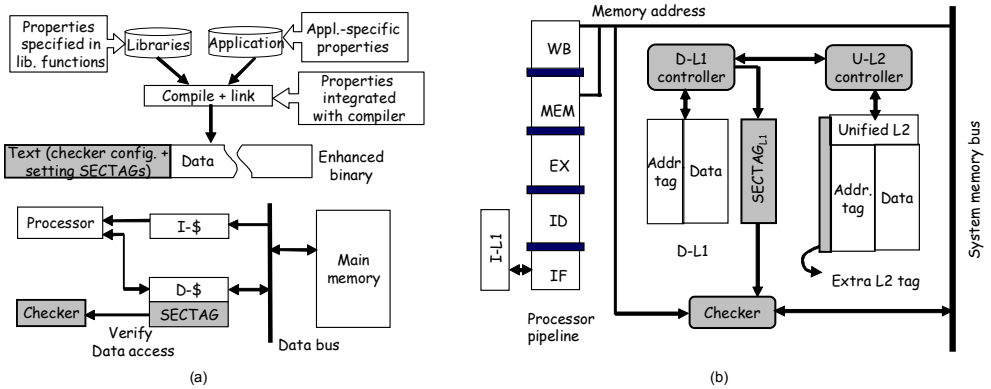

Fig.4. Run-time data validation: (a) Proposed framework (b) Architecture

Fig.4(a) shows the overall design flow for the proposed framework. We augment a program's abstract data state with a new field – SECTAG (SECurity TAG) that encodes its security-specific attributes. These attributes may be derived from application-specific policies, *e.g.*, allowing only some functions to access specified data structures, or they may be universally applicable, *e.g.*, disallowing writes to unallocated memory. The former require modification to application source code while the latter may be implemented with help from run-time libraries or the compiler. The flow produces an enhanced binary, which is run on the enhanced processor that has extra storage and control to manage SECTAGs of data and a dedicated hardware checker to check these SECTAGs and validate data accesses.

Fig. 4(b) shows the details of the architecture with new/modified parts highlighted. We assume a cache configuration with unified L2 cache and split Instruction and Data L1 (I-L1 and D-L1) caches. Different design considerations guided this architecture. In L1 design, to minimize impact on L1 hit time, extra storage - $SECTAG_{L1}$ - is added on-chip to store SECTAGs of data in D-L1. When the processor accesses a datum in D-L1, the checker accesses its SECTAG in parallel and verifies if the data access is allowed by the semantics of the SECTAG. For L2, a design similar to above is infeasible as most processors have large unified L2 caches, and having a separate tag area for each cache line would lead to large hardware overheads. Therefore, we designate lines in L2 to store SECTAGs of other L2 lines and modify cache tags to maintain bookkeeping information about where in L2 the SECTAGs of a particular data line are kept. Cache hit/miss handling policies are modified to fetch SECTAGs from memory and manage them during execution.

The checker is a programmer-visible peripheral that can be configured via registers. For example, the program can specify the address range of data to be checked, exact SECTAG value to be matched on every read/write, SECTAG bound to be compared against on every access, *etc*. Lastly, the instruction set is augmented with and extra instruction *(tsb $rs, $rt)* that sets the SECTAG of the address in *$rs* to the value in *$rt*. This framework is capable of supporting a wide range of security policies including preventing reads from uninitialized memory, segregating data into security levels and restricting access by functions belonging

to different parts of a program, restricting access to stack variables of a function by its callees, *etc.*

5. Experiment and Results

To evaluate the proposed techniques, applications were selected from the embedded benchmark suites, Mediabench and Mibench benchmark suites. Cycle-accurate simulations were performed by modifying Simplescalar [17] to simulate the enhanced architectures.

5.1. Hardware-assisted Control Flow Monitoring

We performed area estimation by synthesizing the monitors for several applications at different granularities. With a base case of ARM920T 32-bit processor core, the average area overheads for inter-procedural, intra-procedural and instruction integrity checking were 0.72%, 2.44%, and 5.59% respectively.

Performance estimation also revealed minimal overheads – the penalty was <1% for inter-procedural checking, 1.77% for intra-procedural checking in *detection mode* (when the processor is allowed to continue while the monitor is checking and stalled only on a control instruction), and 4.94% for intra-procedural checking in *prevention mode* (no new instruction is permitted to commit before checking completes). Instruction integrity checking usually results in substantial overhead, but it can be virtually eliminated by using a pipelined hash engine.

5.2. Hardware-assisted Validation of Program Data Properties

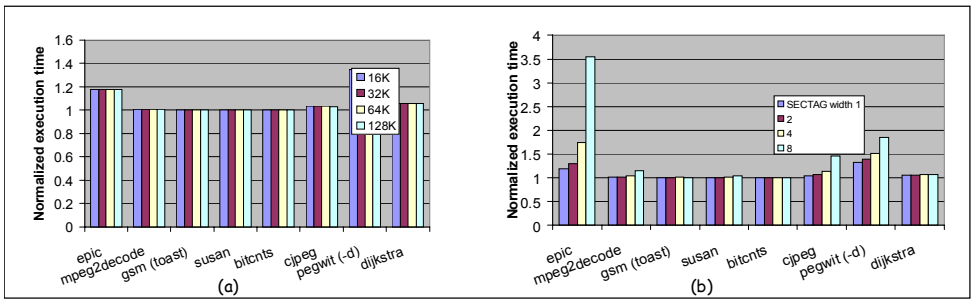

Fig.5. Performance penalty for checking heap accesses with (a) varying L2 sizes, and (b) varying SECTAG bit-widths

Fig. 5(a) shows the performance impact for implementation of the policy: disallow reads/ writes to unallocated memory. Library routines *malloc()* and *free()* are modified to maintain a 1-bit SECTAG for all allocated memory. This policy catches heap overruns, access to freed memory, after it has been freed and exploits that operate by corrupting the *malloc* chunk header. The average and maximum execution time penalty for the default L2 cache size (64K) are 7.6% and 32.7%. Fig. 5(b) shows the scalability of this approach in

supporting higher bit-width SECTAGs. The incremental performance loss to go from a single-bit tag to widths of 2, 4 and 8 is 2.6%, 11.2% and 43.5% respectively.

6. Conclusions

Ensuring secure execution of software is a critical aspect of modern embedded systems. This paper presented two techniques for ensuring secure program execution. The techniques exploit a combination of hardware and software modifications to achieve this objective, by monitoring pre-defined control and data properties. Experimental results indicate that the proposed modifications are reasonable in terms of the overheads incurred, and ensure that a wide range of security attacks can be prevented.

7. References

[1] IBM, Global Business Security Index Report. http://www.ibm.com, 2005.

[2] P. Kocher, R. Lee, G. McGraw, A. Raghunathan and S. Ravi, "Security as a New Dimension in Embedded System Design," in *Proc. ACM/IEEE Design Automation Conference*, pp. 753-760, June 2004.

[3] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," in *ACM Trans. on Embedded Computing Sys*, pp. 461-491, Aug. 2004

[4] Secure Mobile Systems, Mobile Threats. http://security.fb-4.com/mobilealerts.html

[5] R. York, A New Foundation for CPU Systems Security, ARM Limited, 2003 (Available at http://www.arm.com/armtech/TrustZone?OpenDocument.)

[6] Texas Instruments Inc., OMAP Platform(Available at http://focus.ti.com/omap/docs/omaphomepage.tsp)

[7] *MP21x mobile application processors*. NEC Electronics Corp. (Available at http://www.necel.com/en/ techhighlights /application_processor/).

[8] M. Kanellos, *AMD, Intel put antivirus tech into chips*. CNET Networks Inc. (http://news.zdnet.com/2100-1009_22-5137832.html?tag=nl).

[9] C. Cowan et al., "Stackguard: Automatic adaptive detection and prevention of bufferoverflow attacks," in *Proc. USENIX Security Symp*., Jan. 1998, pp. 63–77.

[10] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur, "Run-time detection of heap-based overflows", in *Proc. USENIX Large Installation Systems Administration Conf.*, pp. 51- 60, Oct. 2003.

[11] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *Proc. USENIX Security Symp*., pp. 191-206, Aug. 2002

[12] S. H. Yong and S. Horwitz., "Protecting C programs from attacks via invalid pointer dereferences," in *Proc. European Software Engineering Conf.*, pp. 307-316, Sept. 2003

[13] 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. on Architectural Support for Programming Languages and Operating Systems*, pp. 168–177, Nov. 2000

[14] 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. on Supercomputing*, pp. 160–171, June 2003

[15] D. Arora, S. Ravi, A. Raghunathan and N. K. Jha, "Secure Embedded Processing through Hardware-assisted Runtime Monitoring", *in Proc. ACM/IEEE Design, Automation, and Test in Europe (DATE)*, March 2005

[16] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha, "Enhancing Security Through Hardware-assisted Run-time Validation of Program Data Properties", in *Proc. ACM/IEEE International Conference on Hardware Software Co-design and System Synthesis (CODES+ISSS)*, Sept. 2005

[17] D. Burger and T.M. Austin, "The simplescalar toolset, Version 2.0," Comp. Sciences Dept, UW, Tech. Rep., June 1997.