A Framework for Defending Embedded Systems Against Software Attacks

NAJWA AARAJ, Princeton University
ANAND RAGHUNATHAN, Purdue University
NIRAJ K. JHA, Princeton University

The incidence of malicious code and software vulnerability exploits on embedded platforms is constantly on the rise. Yet, little effort is being devoted to combating such threats to embedded systems. Moreover, adapting security approaches designed for general-purpose systems generally fails because of the limited processing capabilities of their embedded counterparts.

In this work, we evaluate a malware and software vulnerability exploit defense framework for embedded systems. The proposed framework extends our prior work, which defines two isolated execution environments: a *testing* environment, wherein an untrusted application is first tested using dynamic binary instrumentation (DBI), and a *real* environment, wherein a program is monitored at runtime using an extracted behavioral model, along with a continuous learning process. We present a suite of software and hardware optimizations to reduce the overheads induced by the defense framework on embedded systems. Software optimizations include the usage of static analysis, complemented with DBI in the testing environment (i.e., a hybrid software analysis approach is used). Hardware optimizations exploit parallel processing capabilities of multiprocessor systems-on-chip.

We have evaluated the defense framework and proposed optimizations on the ARM-Linux operating system. Experiments demonstrate that our framework achieves a high coverage of considered security threats, with acceptable performance penalties (the average execution time of applications goes up to 1.68X, considering all optimizations, which is much smaller than the 2.72X performance penalty when no optimizations are used).

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design

General Terms: Security, Design, Performance

Additional Key Words and Phrases: Behavioral analysis, embedded systems, instrumentation, malware, multiprocessor systems, software vulnerabilities

ACM Reference Format:
Aaraj, N., Raghunathan, A., and Jha, N. K. 2011. A framework for defending embedded systems against software attacks. ACM Trans. Embedd. Comput. Syst. 10, 3, Article 33 (April 2011), 23 pages.
DOI = 10.1145/1952522.1952526 http://doi.acm.org/10.1145/1952522.1952526

1. INTRODUCTION

Embedded systems, which nowadays account for a majority of electronic systems, are increasingly being deployed in physically insecure spaces. As these systems begin to

This work was supported by NSF under Grant No. CNS-0720110.
Authors' current addresses: N. Aaraj, Booz-Allen-Hamilton, 812, Tabaris, Charles Malek Avenue, P.O. Box 16-6541, Beirut, Lebanon; email: najwa.aaraj@booz.com; A. Raghunathan, School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Ave., West Lafayette, IN 47907-2035; email: raghunathan@purdue.edu; N. K. Jha Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; email: jha@ee.princeton.edu.

DOI 10.1145/1952522.1952526 http://doi.acm.org/10.1145/1952522.1952526

feature significant complexity and the ability to connect to public networks and download programs from the Internet, a new threat dimension that researchers should be on the lookout for is the exploitation of inherent vulnerabilities in embedded software and the spreading of malicious code on such systems. Examples include the spread of viruses and malware to mobile phones, cars, and consumer appliances. On the other hand, embedded systems pose unique challenges for "security processing" [Ravi et al. 2004], that is, the computational time required for the purpose of security. Our experience with adapting general-purpose security solutions to embedded systems indicates that the computational demands of such solutions can easily overwhelm the embedded system processing capabilities. Therefore, building security solutions directly tailored to embedded systems is an important research front.

In this work, we address the problem of defending embedded platforms against software exploits. As embedded systems feature a continuous increase in their software content, most security attacks on them are software-based [Cabir 2004; Flexispy 2006] and occur either by exploiting flaws in a program that is not malicious itself or through the execution of malicious code. We propose an efficient framework to address software exploits in unknown binary applications whose source code is not available, by using several important techniques.

We build our embedded framework on top of the system proposed in Aaraj et al. [2008], which addresses the problem of defending against software exploits on general-purpose systems. It is based on observing the execution of a program, whose source code is not available, modeling safe and unsafe behavior with respect to specified postexecution security policies, and ensuring that a program does not deviate from safe behavior. The approach utilizes the concept of isolated execution environments and dynamic binary instrumentation (DBI)—DBI is a technique which injects code into an application that allows the observation of an application's behavior at various execution stages. It defines (1) a *testing* environment (i.e., a duplicate of the real user workspace), where the behavior of an unknown program, running under DBI, is observed and analyzed, postexecution, against a set of security policies that models safe/unsafe behaviors; and (2) a *real* environment (i.e., the real user workspace), where a program is monitored at runtime using a behavioral model B_M, extracted in the testing environment as a result of applying the security policies, along with a continuous learning process.

We have extended the above-described approach to embedded systems by implementing the Testing environment using an embedded system emulator running on a remote server or desktop—this allows us to safely test an untrusted embedded application without the danger of corrupting the "live" environment (i.e., the real environment or the real embedded platform) or overwhelming it with high computational requirements. Our work makes the following contributions.

—We perform a comprehensive characterization of the defense framework in the real environment in order to identify performance hotspots. We also measure the increase in execution time as a function of B_M's blocks, which are used for runtime monitoring in this environment.
—We propose a set of software optimizations in the testing environment to reduce the run-time overhead on the real embedded platform. Software optimizations consist of the following.
  (1) Before running a program under DBI and analyzing its behavior in the testing environment, we statically disassemble its binary, and identify *safe* and *potentially unsafe* code segments within the program's recovered machine code instructions, using a set of restrictive static policies. As a result, when the program is tested, only *potentially unsafe* code segments are subjected to DBI.

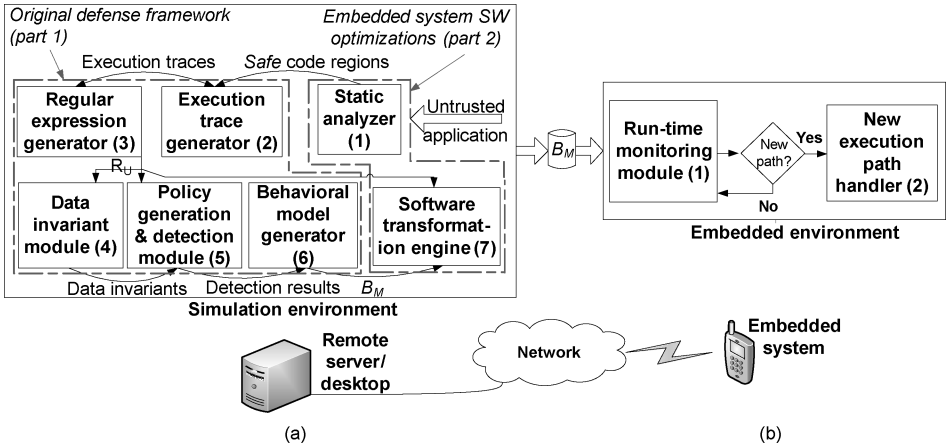

Fig. 1. Framework overview for the (a) testing and (b) real environments.

(2) We reduce the number of blocks in B_M by performing several software optimizations consisting of dead-code elimination. As our computational analysis shows, the execution time in the real environment increases with the number of blocks in B_M.

—We propose a set of hardware optimizations for multiprocessor systems. We identify coarse-grained parallelism between different tasks running in the real environment in order to exploit the parallelism provided by multiprocessor systems-on-chip (SoCs).

—We perform experiments on a large set of open-source and synthetic benchmarks. The testing environment is implemented on an ARM processor simulated using the QEMU system emulator [Qemu 2008]. Experiments in the real environment are evaluated on a battery-powered handheld device (Sharp Zaurus PDA). Experimental results include execution time and energy analysis in the real environment, and software exploit detection and prevention rates in the testing and real environments.

The rest of the article is organized as follows. An overview of the proposed framework is given in Section 2. A detailed analysis of the framework environments is given in Section 3. Details of the software optimizations are presented in Section 4. The multiprocessor optimization is presented in Section 5. Experimental results for the original framework and respective software and hardware optimizations are given in Section 6. A survey of relevant past work is presented in Section 7. Our future work is discussed in Section 8, and our conclusions given in Section 9.

2. FRAMEWORK OVERVIEW

Figure 1 presents the architecture of the defense framework in both execution environments. Before being deployed on the actual embedded device, an application is first tested in the simulated embedded environment (testing environment) running on a desktop system. The application's behavioral model, B_M, is consequently extracted and wirelessly transmitted to embedded devices, where runtime monitoring against B_M is performed each time the application is executed. Figure 1(a) presents the defense framework originally proposed in Aaraj et al [2008] (part 1) and the software components added for optimizing the framework on embedded systems (part 2).

The static analyzer (block 1 in Figure 1(a)), added as a software optimization component, identifies *safe* and *potentially unsafe* code regions in the restored machine code instructions of the tested application, based on designed static security policies. The

execution trace generator (block 2 in Figure 1(a)) executes the untrusted program under a series of automatically or manually generated inputs. While building the program execution traces, DBI, based on the Pin [Hazelwood and Klauser 2006] framework, is used to intercept execution within statically identified *potentially unsafe* regions and generate information (control and data) that is necessary for analyzing the application's execution events.

The regular expression generator (block 3 in Figure 1(a)) transforms each sequence of instructions terminating in a control flow transfer instruction (i.e., basic block BB_i) into block element bb_i, where bb_i is a one-to-one mapping from BB_i, such that bb_i contains data component C_j if instruction I_j in BB_i executes action A_j. For instance, if instruction I_0 in basic block BB_i calls functions such as *memset*, *memcpy*, etc., we define data component $C_0 = copy\_fct$ in the corresponding bb_i. C_0 contains the size of the data to copy, and the starting address where data are copied, that is, the allocated memory space.

Each execution trace is subsequently expressed as a regular expression R_k defined over alphabet $\Sigma = \{bb_1, \ldots, bb_n\}$. A recursive union of all R_k's is performed to combine all execution traces into a single regular expression, R_U. Meanwhile, regular expressions are passed through a data invariant module (block 4 in Figure 1(a)), which formulates invariants obeyed by the data associated with each block element bb_i. Invariants are used to characterize the properties assumed by the different data components, C_j's, of bb_i.

R_U is then subjected (through regular expression intersection) to postexecution security policies, P_i's, in order to detect any software security exploit (block 5 in Figure 1(a)). P_i's are a regular expression-based translation of a high-level specification of a series of events, which, if occurring in a particular sequence, imply a security violation. Thereafter, a behavioral model, B_M, is derived from R_U based on the results of applying the security policies (block 6 in Figure 1(a)). B_M is a simplified model, which contains a reduced set of bb_i's. It encapsulates permissible (or nonpermissible) real-time behavior of a program by embedding suitable properties and flags (substitutes of security policies) within its blocks. B_M is further optimized by the software transformation engine (block 7 in Figure 1(a)), and then migrated to the real environment.

In the real environment, the running application's basic blocks, which match the blocks of B_M, are subjected to instrumentation and runtime monitoring (block 1 in Figure 1(b)). Runtime monitoring also involves the application of restrictive security policies in case new execution paths are encountered (block 2 in Figure 1(b)).

3. ANALYSIS OF THE DEFENSE FRAMEWORK

This section first gives a more detailed explanation of the original defense framework presented in Aaraj et al. [2008] (part 1 in Figure 1(a)). It then explains the motivation behind the software optimizations (part 2 in Figure 1(a)), targeted toward reduced overheads on embedded platforms.

3.1. Original Defense Framework Reviewed

This section gives a brief explanation of the different components in the testing and real environments.

3.1.1. Testing Environment Components.

3.1.1.1. Execution Trace and Regular Expression Generators. Our tool is built on top of Pin [Hazelwood and Klauser 2006], a DBI framework from Intel. It executes instructions intercepted and regenerated by Pin and generates information that is used to check for software exploits. For example, in case we are targeting the detection

of a buffer overflow vulnerability exploited to redirect a function pointer to execute attack code, our tool generates the following information: (1) memory allocation size and starting address; (2) writes to allocated memory through functions such as *memcpy*, *sprintf*, *memset*, etc., and (3) value of function pointer address.

As an application executes, our tool monitors its control flow and a regular expression R_k is built to represent an execution instance of the application. While building R_k, we consider basic block BB_i as the unit of execution. BB_i is a set of sequentially executed instructions exiting at a control transfer instruction. R_k is defined over alphabet $\Sigma = \{bb_1, \ldots, bb_n\}$, where bb_i is a one-to-one mapping from BB_i, using the following homomorphism, h_s: $BB \rightarrow bb$; $\forall\ BB_i$, $h_s(BB_i) = bb_i$, such that bb_i contains data component C_j if instruction I_j in BB_i executes action A_j. Using the same function pointer example mentioned earlier, if A_0, corresponding to I_0 in BB_i, executes memory allocation, then the corresponding data component $C_0 = alloc\_mem$ in bb_i, contains the starting address (S_A) and size of allocated memory (n_A). Similarly, for A_0 in $BB_{i+\gamma}$ that calls functions such as *memset*, *memcpy*, etc., we define $C_0 = copy\_fct$ in $bb_{i+\gamma}$, which contains the size of the data to copy (n_D), and the starting address (S_A) where the data is copied, that is, the written-to memory space. For A_0 in $BB_{i+\delta}$, implementing a function pointer call, we define $C_0 = function\_ptr$, containing the address of the function pointer (F_A) at invocation time. For a complete discussion of the generation of regular expression R_k, its blocks, and associated data components, we refer interested readers to Aaraj et al. [2008].

The precision and completeness of our framework depends on the number of tested feasible paths in the testing environment. Using a set of input sequences, we generate an execution regular expression, R_k, at each execution instance of the untrusted program. Regular expressions are recursively subjected to a union operation, in order to generate the final execution regular expression, R_U. A prototype implementation of $R_k \cup R_U$ can be found in Aaraj et al. [2008].

3.1.1.2. Data-Invariant Module. In this section, we use the term *invariants* to characterize the properties assumed by the different data components, C_i's, of bb_i. Properties assumed by the different C_i's can assume any of the following invariant types: (1) acceptable or unacceptable constant values (i.e., a value that a variable should or should not assume, respectively), (2) acceptable or unacceptable range limits (i.e., the minimum and maximum values that a variable can assume or should not assume, respectively), (3) acceptable or unacceptable value sets (i.e., the set of values that a variable can assume or should not assume, respectively), or (4) acceptable or unacceptable functional invariants (i.e., a relationship among a number of variables that should or should not be satisfied, respectively). The determination of which invariant category data component C_i assumes is based on the observations of this component over different executions of the untrusted application. We follow the methodology described in Perkins and Ernst [2004] to maintain single or multiple invariant categories for each data component C_i. A more detailed explanation on deriving data invariants can be found in Aaraj et al. [2008].

Invariants are used by the detection module, where they are checked against designed postexecution security policies. They are also embedded within the blocks of behavioral model B_M to indicate permissible or nonpermissible runtime behavior in the real environment, as explained next.

3.1.1.3. Security Policies and Detection Module. After deriving the final regular expression R_U and its associated C_i's and data invariants, in order to detect any software exploits, we use different security policies that identify an anomalous exploit in program execution. Policies, P_i's, are a translation of a high-level specification of a series

of events, which, if occurring in a particular sequence, imply a security violation. Equation (2) shows the structure of a security policy P_i and its high-level specification, H_i (Equation (1)).

$$H_i = \{\textbf{if } Action_1 \textbf{ then} \ldots \textbf{ then } Action_m \Rightarrow \textbf{Action A} \quad (1)$$

$$P_i = [b_1.(b_k, k \neq 2, \ldots, m)^*]^*.[\ldots]^*.b_m \Rightarrow \textbf{A} \quad (2)$$

For example, in order to detect a buffer overflow exploit targeting function pointer values, we design the following security policy P_0 (Equation (4)), corresponding to specification H_0 (Equation (3)).

$$H_0 = \begin{cases} A\ call\ to\ a\ function\ pointer\ following\ an\ illegitimate\ overwrite\ of\ the \\ function\ pointer \Rightarrow \textbf{Software exploit} \end{cases} \quad (3)$$

Illegitimate overwrites are detected as calls to any of several intercepted functions, for example, *memcpy, sprintf, memset,* etc., that change the value of the function pointer even though the address scope of their arguments does not include the function pointer address.

$$P_0 = [b_2.(b_k, k \neq 1)^*]^*.b_1 \Rightarrow \textbf{Software exploit} \quad (4)$$

$\boldsymbol{b}_1$ is a block with C_i corresponding to a function that can induce a buffer overflow of allocated memory, corrupting the value of the function pointer, and $\boldsymbol{b}_2$ a block with C_i corresponding to a function pointer with an overwritten address value (e.g., $C_0 = copy\_fct$ in $bb_{i+\gamma}$, $C_0 = function\_ptr$ in $bb_{i+\delta}$, and $n_D > |S_A - F_A|$).

If the sequence of blocks b_i in a security policy is captured in R_U, through a regular expression intersection operation between R_U and P_i, a software exploit must have occurred during the application's execution.

3.1.1.4. Behavioral Model Generation. Behavioral model B_M is a simplified model derived from R_U based on applying the set of designed security policies P_i's. It efficiently reflects the intended program behavior, while eliminating any bb_i, which contributes redundant information for runtime monitoring. Blocks bb_i's of R_U that are included in B_M are as follows.

(1) *Anomaly-initiating (AI) blocks.* Such blocks in B_M are the first blocks of an execution sequence that might be malicious with respect to a security policy P_i (e.g., blocks that contain memory allocation components).
(2) *Anomaly-dependent (AD) blocks.* Such blocks in B_M are correlated with previous *AI* or *AD* blocks (e.g., blocks that contain calls to functions such as *memcpy, sprintf, memset,* etc., that can result in a buffer overflow when invoked).
(3) *Anomaly-concluding (AC) blocks.* Such blocks in B_M are correlated with previous *AI* and *AD* blocks, and their execution determines whether an execution sequence is safe or not with respect to a security policy P_i (e.g., blocks that contain a call to a function pointer).
(4) *Conditional blocks:* Such blocks in B_M are maintained in order to keep track of observed feasible paths of the untrusted binary in the testing environment.

In addition to the simplified set of blocks, B_M encapsulates, within its blocks, permissible (or nonpermissible) real-time behavior of executing programs. This behavior is enforced by embedding suitable properties and flags (substitutes of security policies) within B_M's blocks. For example, flag ***Fct_Ptr_Verify*** is added to bb_i of B_M, which contains element C_i corresponding to a function pointer call after a write operation to allocated memory. This flag forces the runtime monitor to check the value of the

function pointer Fct_Ptr against invariant: Inv_1: $Fct\_Ptr = C$ (C is a constant value, indicating a permissible behavior). If the invariant is satisfied at runtime, the application is allowed to execute. Otherwise, execution is halted.

3.1.2. Real Environment Components.

3.1.2.1. Runtime Monitoring Based on Model B_M. Real-time monitoring starts by loading the application's behavioral model B_M that is migrated from the testing environment. During program execution, the monitor automatically delineates a basic block (denoted BB_{real}) and compares it against the sofar scanned bb in B_M (denoted bb_M). If the two blocks do not match, the instrumentation of BB_{real} is suspended until a new block is identified. Otherwise, the monitor instruments the instructions within BB_{real} until its end is identified. Each instruction is checked against properties and invariants of bb_M. Once checks are cleared, instructions within BB_{real} are allowed to execute and commit; otherwise the program's execution is suspended. The checks performed at each instrumented instruction are based on the information stored at bb_M. For the function pointer call example, we check the value of the function pointer against the value of the invariant stored in the ***Fct_Ptr_Verify***-flagged bb_M.

If BB_{real} contains a conditional control transfer, and a new execution path is observed, we proceed as explained next.

3.1.2.2. Handling New Execution Paths. In case a new feasible path is observed, the monitor aims at preventing any software attack in the newly executed path. The new path instrumentation proceeds until the exit of the conditional path is identified (a condition's exit point is stored as a block in B_M). At that point, real-time monitoring against B_M resumes. During instrumentation of the new path, a set of highly-restrictive policies is enforced at runtime before any instruction completes execution. While those policies may eventually result in some false positives, they keep the overhead in the real environment low and prevent the execution of malicious code. Basic blocks in newly executed paths, to which the restrictive security policies were applied, and which did not result in halting of execution, are added to behavioral model B_M.

Restrictive policies are based on the following observations:

(1) *AC* blocks of a newly observed feasible execution sequence (i.e., a feasible execution sequence that was not observed in the testing environment), which are correlated to *AI* and *AD* blocks already observed in the testing environment, are allowed to execute if and only if specific conditions are satisfied.
(2) *AC* blocks of a newly observed feasible execution, which are correlated to *AI* and *AD* blocks not observed in the testing environment, are not allowed to execute.

For example, a highly restrictive policy does not allow the execution of a function pointer call if calling the function pointer follows a write to allocated memory.

3.2. Computational Characteristics of the Original Defense Framework

We next analyze the computational characteristics of the original defense framework on an ARM926EJ-S embedded processor simulated using the QEMU emulator [Qemu 2008].

The callgraph associated with the defense framework in the real environment is shown in Figure 2. It reveals that monitoring the control flow against B_M ($Build\_CFG$) takes around 68.37% of the execution time, and 27.58% of the execution time is spent in $Build\_DFG$ (monitoring the data flow as dictated by flags and invariants embedded in B_M's blocks). On the other hand, Equation (5) presents a macromodel that captures the execution time ratio (ex_ratio) in the real environment as a function of the number

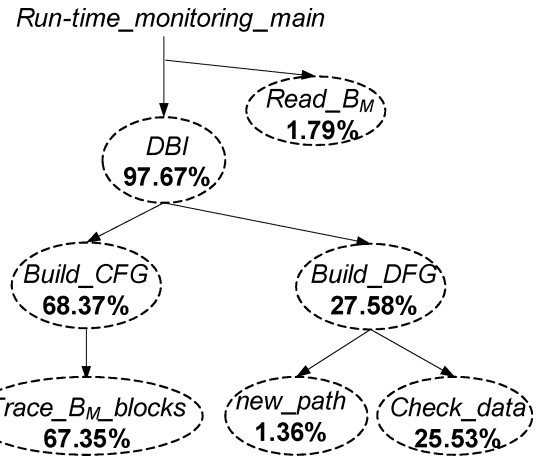

Fig. 2. Callgraph for runtime monitoring in the real environment.

of blocks in B_M [nb = (number of block in B_M)$\times 10^{-3}$]. As can be concluded from the macromodel, execution time increases with nb.

$$ex\_ratio = -0.075 \times (nb)^2 + 0.793 \times (nb) + 1.044. \quad (5)$$

As discussed before, blocks that constitute B_M are the *AI, AD, AC*, and *Conditional blocks*.

We make the following observations.

(1) If sequence (AI, $AD_0, \ldots, AD_k$, AC) can never result in a malicious software exploit, it does not need to be included in the behavioral model. Such sequences can be detected by preceding *DBI* with static analysis of the disassembled untrusted binary through subjecting its recovered machine code to a set of static security policies. Static analysis (static analyzer in Figure 1(a), part 2) is performed before generating execution traces and is explained in detail in Section 4.1.
(2) If the controlling expression of a conditional block always results in the execution of the same path, this conditional block does not need to be included in the set of blocks in B_M. The software transformations guarantee the exclusion of such conditional blocks from B_M. Software transformations (Figure 1(a), part 2) are performed after the generation of behavioral model B_M and is explained in detail in Section 4.2.

4. SOFTWARE OPTIMIZATIONS

This section discusses the software optimization components proposed for an efficient deployment of the defense framework on embedded systems.

4.1. Static Analyzer

The main rationale of this step is to reduce the number of AI, AD_i, and AC blocks in B_M, thus, reducing nb and, consequently, the execution time ratio (ex_ratio) in the real environment.

In the static analysis step, we disassemble the binary of the untrusted application. Disassembly recovers the application's machine code instructions from its binary. The analysis method described below is orthogonal to the disassembler being used. However, using disassemblers least vulnerable to code obfuscations [Kruegel et al. 2004] is preferable.

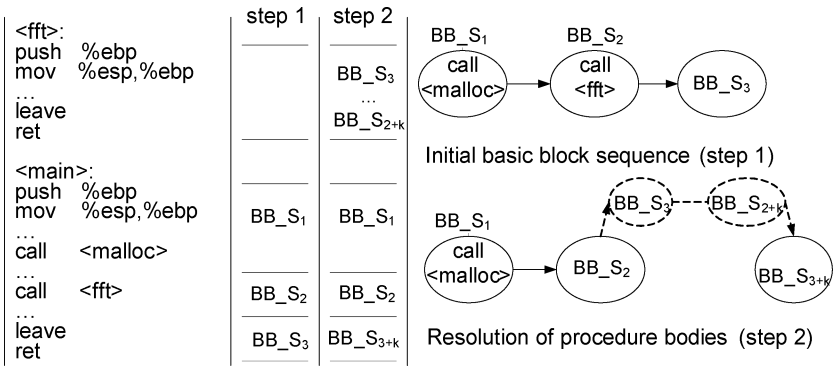

Fig. 3. Extraction of CFG in the form of a basic block sequence.

Based on the application's recovered machine code, we construct its control flow graph (CFG) as a sequence of basic blocks (BB_S_i). We then generate a functionally equivalent regular expression S. S is defined over alphabet $\sigma = \{bs_1, \ldots, bs_n\}$, whereas bs_i contains statically-extracted functional and data information.

4.1.1. Building CFG as a Basic Block Sequence. We construct the CFG of an application as a sequence of basic blocks (BB_S_i) executed successively. A straightforward approach to building a basic block is to mark a control transfer instruction as the block's exit point. The main difficulty in building the CFG is correctly accounting for the executed code of a called procedure. To handle this problem, we replace procedure bodies within their calling context in order to specialize them to particular usage patterns. We proceed in two steps. In the first step, an initial sequence of basic blocks is constructed. The second step recursively replaces procedure bodies within the basic blocks of the initially derived sequence.

4.1.1.1. Initial Basic Block Sequence. In order to derive the correct sequence of basic blocks that are likely to be executed at runtime, we separate basic blocks at the *main* function entry point. Instructions are grouped within a single block until a control transfer instruction is encountered. The basic block identification process is continued at all local direct jump targets. For local indirect jumps, both corresponding branches are grouped as possible successor basic blocks. In case a function call instruction is encountered, the process continues at the address following the instruction. Basic block identification continues until *main*'s exit point is identified.

4.1.1.2. Resolution of Procedure Bodies. The aim of this step is to resolve function calls at basic blocks, BB_S_i, derived in the initial sequence. It proceeds by fetching the body instructions of each procedure called in *main* within the disassembled binary (if available). Fetched instructions are grouped in basic blocks, and added as successors of locally identified blocks from within which they are called. If instructions of a called function are not available within the disassembled binary, the process of identifying basic blocks continues at the address following the function call instruction.

This process of resolving procedural bodies is recursively continued until the instructions of all invoked function calls (within *main* or within functions called by *main*) are grouped into corresponding basic blocks and correctly placed in the program execution flow.

The recursive approach used to identify the correct sequence of executed basic blocks is illustrated in Figure 3 and given in Algorithm 1.

ALGORITHM 1: build_CFG algorithm

Input: start_point where to start building the CFG
Output: array ins = statically built CFG

```
function <build_CFG>(<start_point>) {
    instruction = start_point.begin();
    index = 0;
    while(instruction not equal end_function) {
      if (instruction.operand == call) {
        if (called_function.expand == True) {
          compare(called_function.calls_array, start_point.is_called_array);
    // Avoiding infinite loops
          if match
            ins[index++] = instruction;
          else
            build_CFG(called_function);
        }
        else
          ins[index++] = instruction;
      }
      else
        ins[index++] = instruction;
      instruction ++;
    }
    return;
}
```

4.1.2. Building S and Interbasic Block Slicing. We build S as a sequence of blocks bs, where each bs_i is a one-to-one mapping to basic block BB_S_i. When BB_S_i is identified, we scan it for instructions that might eventually contribute to a software exploit (i.e., any actions that can be performed within an *AI*, *AD*, or *AC* block). Such instructions are added as functional components fc_k's to bs_i.

At this point, we are concerned with the problem of interbasic block slicing, that is, generating a slice of the untrusted application, where each slice crosses the basic block boundaries. An interbasic block slice with respect to a program point p in BB_S_i, where a functional component is identified, consists of backward-traversed instructions of BB_S_x with $x < i$, whose operands might affect the value of the arguments of fc_k's in bs_i. In case we can concretely define, at instruction I_m, the value of fc_k's arguments at point p, and in case there are no intervening definitions of those arguments in the execution path between I_m and p, such values are added as data components dc_{kj}, corresponding to fc_k in bs_i.

We compute interbasic block slices using backward dataflow analysis on recovered machine code instructions, where each instruction is replaced with the appropriate sequence of assignments to procedure inputs, in case such assignments exist.

An example bs_i and its corresponding functional and data components are given next for the function pointer example discussed in previous sections. As shown in Table I, instruction I_k has the potential of inducing a buffer overflow attack if the size of copied data is larger than the size of the allocated memory space at instruction I_f. Therefore, *memset* is added as functional component fc_1 in bs_1, and instructions I_j through I_e are backtracked or reverse-traversed to determine the arguments of *memset*, added as dc_{11} and dc_{12} to bs_1.

Similarly, I_m contains a function pointer call, which is added as fc_1 of corresponding block bs_2. As the function pointer value cannot be determined at compile-time, no data elements are added to bs_2.

Table I. Extracted Functional and Data Components for Function Pointer Call Example

Recovered instruction	fc_k	dc_{kj}
I_k: call <memset@plt> I_j: push \$0×3 I_i: pushl **0xfff4(%ebp)**	fc_1 (bs_1): *memset*	dc_{11}: size of copied data = 3
I_e: push \$0×4 I_f: call <malloc@plt> I_g: add \$0×10,%esp I_h: mov %eax, **0xfff4(%ebp)**		dc_{12}: size of allocated memory space = 4
I_m: call *%eax	fc_1 (bs_2): function pointer call	dc_{11}: none

Table II. Exploit-Specific Functional and Data Components in bs_i

	Functional component fc_k	Data component dc_{kj}
R_1	fc_i = *memset, memcpy, etc.*	dc_{i1} = size of written data
		dc_{i2} = size of allocated memory
R_2	fc_i = function return	dc_{i1} = none
R_3	fc_i = function pointer call	dc_{i1} = none
R_4	fc_i = call to "exec"	dc_{i1} = none
R_5	fc_i = GOT function call	dc_{i1} = none
R_6	fc_i = sys_call$_1$	dc_{i1} = none
R_7	fc_i = sys_call$_2$	dc_{i1} = none

Other examples of extracted exploit-specific functional and data components are listed in Table II. Examples in the table correspond to vulnerability exploits, such as buffer overflows and race conditions.

(1) *Buffer overflows.* Buffer overflows are responsible for a major portion of security vulnerabilities in embedded software. It was shown in Wilander and Kamkar [2003] that various attack targets (return address, old base pointer, function pointers, longjmp buffers, and "exec" arguments) can be exploited by buffer overflows to change the control flow of a program.

Buffer overflow attacks are also implemented as noncontrol data attacks, which operate by overflowing the value of noncontrol application data, such as system call arguments, user identity data, global offset table (GOT) entries, configuration data, etc. [Chen et al. 2005].

Detecting buffer overflows statically is very inaccurate without modification or recompilation of the source code [Larochelle and Evans 2001]. However, we can identify code sections that are *safe* from buffer overflow exploits if the sizes of both the data-to-copy and the allocated memory space are defined at compile time. Hence, we specify safe regions in the disassembled binary with respect to buffer overflow attacks according to the following observation.

If it can be determined at compile time that writing n_1 data bytes to a buffer space of size n_2 bytes always satisfies $n_1 \leq n_2$, then the corresponding code section is statically identified as safe.

Functional and data components derived to identify such safe regions are given in rows R_1–R_5 of Table II.

Note that static security policies are defined for buffer overflow targets that can be resolved at compile-time, that is, function returns, function pointers, "exec" arguments, and GOT entries.

(2) *Race conditions*. Race conditions are a common form of vulnerabilities and the exploit of security holes related to them enables an attacker to gain read, write, and execute privileges that interfere with the file system. We classify race conditions into three categories.

— *(sys_call$_1$-sys_call$_2$) attack.* The attacker exploits a time window between two system calls operating on a particular file F. Most frequently, such an attack

is accomplished through the creation of a symbolic or hard link from F to a privileged file. Most common (sys_call_1-sys_call_2) pairs are: (access-open), (open-open), (open-chmod), (rename-chown), (fstat-open), (chdir-rmdir), and (mkdir-chmod).

—*Directory redirection attack.* The attacker redirects the directory, where an event was being performed, to another directory.

—*Temporary-file attack.* A program creates temporary files with easily predictable names. An attacker exploits this vulnerability by launching link attacks on guessed file names, potentially gaining elevated access privileges to the system.

Flagging safe code regions with respect to race condition attacks can only be done for the first category, where the attacker exploits a time window between two system calls operating on file F (most frequently, such an attack is accomplished through the creation of a symbolic or hard link from F to a privileged file). This attack can be associated with a basic block sequence (AI, AC). If it can be determined statically that sys_call_1 in AI is not followed by sys_call_2 in AC, the race condition attack is unlikely to succeed. Functional and data components derived to identify safe regions with respect to race condition attacks are given in rows R_6–R_7 of Table II.

(3) *Malicious code execution*. Malicious code execution can have various effects on an underlying system, including monopolizing of executables, companion files, loadable kernel modules, kernel source files, etc., through file overwriting, code injection, entry point obscuring, etc. Since these symptoms can also result from executing a benign process or from hardware or software failures, detecting a malicious or benign behavior is very hard to do at compile-time, that is, before analyzing postexecution effects of running the untrusted application. Therefore, we do not design any security policies to statically identify nonmalicious *safe* code segments.

Our implementation of the interbasic block slicing is compiler-specific (the ARM instruction set in particular); however, it can easily be adapted to any compiler. The example shown in Table I is specific to an x86-compiler for ease of illustration.

Generated expression S, with associated functional and data components, is then intersected with a set of static security policies SP_i. Static security policies are designed as a valid sequence (AI, AD_0,. . . , AD_k, AC). A sequence is valid if, when executed, it satisfies particular properties of AI, AD_i, and AC, and cannot result in a software exploit.

Examples of static security policies (corresponding to Table II) are shown in Table III for different types of software exploits.

For each block sequence in S that matches all security policies SP_i, the corresponding code section is flagged as safe. Otherwise, it is flagged as potentially unsafe. Entry point addresses of safe regions are then fed to the *execution trace generator*, which consequently excludes those regions from DBI.

4.2. Software Transformations

This step consists of dead-code elimination and reduces the number of *conditional blocks* in B_M. The conditional blocks that we aim at eliminating include blocks at which one of the conditional targets is never executed (unreachable code), and blocks at which one of the target paths corresponds to code associated with the handling of error conditions.

To address this step, we present a list-based approach, which performs such dead-code elimination using generated individual traces, R_k's, and static analysis.

For all conditional blocks, at which only one target path has been executed in all observed R_k's, we use a heuristic to determine whether the unobserved path cannot be

Table III. Static Security Policy Examples

Software exploit	Static security policy SP_i
Buffer overflow targeting return addresses	$SP_1 = (AI_1, AD_1, AC_1)$ AI_1: allocated memory of size S_M AD_1: block containing call to functions such as *memset*, and n_R (size of bytes to write) $\leq S_M$ AC_1: block containing a function return call
Buffer overflow targeting function pointers	$SP_2 = (AI_1, AD_1, AC_2)$ AC_2: block containing a function pointer call
Buffer overflow targeting "exec" arguments	$SP_3 = (AI_1, AD_1, AC_3)$ AC_3: block containing a call to "exec"
Buffer overflow targeting GOT entries	$SP_4 = (AI_1, AD_1, AC_4)$ AC_4: block containing a GOT entry call
Race condition (sys_call$_1$-sys_call$_2$)	$SP_5 = (AI_5, \neg\, AC_5)$ AI_5: block containing invocation of sys_call$_1$ AC_5: block containing invocation of sys_call$_2$

executed or will only be executed when an error condition arises. This is done by defining a list of condition types of interest and then determining whether the controlling condition of each type leads to the execution of one target path only. Condition types considered in our problem setting include the following:

(1) trigger conditions, such as system call return values, system time, system events, etc;
(2) system configuration settings;
(3) error conditions;
(4) statically evaluated controlling expressions.

Assuming that the untrusted application was tested *num_execution* times in the testing environment, we start by comparing target paths executed at each conditional block in R_k and $R_{k'}$, $\forall\, k, k' < num\_executions$. For all conditional blocks that are always followed by the same executed path, we determine whether the block's controlling condition assumes any of the condition types listed above. Given the condition types of interest, we proceed as follows.

(1) Conditional blocks including trigger conditions are not removed from the scope of B_M. This is particularly important for malicious code that manifests trigger-based behavior.
(2) For controlling expressions related to configuration settings, the unobserved path is ignored since the real environment will have the same settings as the testing environment. Subsequently, the conditional block is removed from within B_M, if the block is not an *AI*, *AD*, or *AC* at the same time.
(3) Whether a controlling expression is related to error handling can be determined from the disassembled binary. In this case, we can remove the associated conditional block from within B_M, as it is very unlikely for an attacker to exploit error handling code.
(4) If the controlling expression can be evaluated at compile time, the elimination algorithm removes the corresponding conditional block in B_M. Determining a controlling condition at compile time is done through constant backward propagation and data flow analysis [Wegman and Zadeck 1991] applied to the application's disassembled binary.

Software optimizations deployed in the testing environment affect B_M as follows. Static analysis reduces the number of *AI*, *AD*, and *AC* blocks by an average of 5.13% (the size of B_M is reduced by 7.52%). Dead-code elimination reduces the number of conditional blocks in B_M by an average of 19.84% (the size of B_M is reduced by an

average of 13.24%). Eliminated blocks mostly contain controlling expressions related to error handling procedures and system configuration settings. Statically evaluated controlling expressions account for only 4.71% of the eliminated blocks.

The combination of both static analysis and dead-code eliminations reduces the number of block in B_M by an average of 24.97%.

As will be seen in Section 6, execution time in the real environment, after implementing the software optimizations, goes up to 2.35X, achieving an execution time reduction of 13.25% as compared to the original defense framework. We note that the software optimizations do not affect the size of the runtime monitor in the real environment, since the latter still needs to check both the control and data flow of the application running in the real environment against B_M, irrespective of the size of the model.

5. OPTIMIZATIONS FOR MULTICORE PLATFORMS

In this section, we present optimizations to further enhance performance on embedded systems that contain multicore processors.

A profile of the function callgraph of runtime monitoring after the implementation of proposed software optimizations assumes almost the same execution time percentages as in Figure 2. Compute-intensive operations in the monitoring process are identified, revealing good candidates to be assigned as parallel tasks, each executing on a separate processor. The latent parallelism of multiprocessor SoCs can be exploited to provide improved execution time of the software running on them.

The challenge of assigning parallel tasks on the multiprocessor system is being able to balance the workload on each processor, while minimizing the processor idle time due to communication between the processors. As can be seen from Figure 2, runtime monitoring in the real environment can be divided into two parallel tasks: the first task (*Build_CFG*) monitors the control flow of the application against behavioral model B_M, and the second task (*Build_DFG*) monitors the data flow against flags and invariants within B_M's blocks. We can clearly observe a large difference between the execution time required by both tasks. However, the overhead incurred by *Build_CFG* is mainly due to the execution of a single task, which is Pin's instrumentation of the application's instructions, constituting about 97.22% of the execution time required by *Build_CFG*. Since intercepting instructions is done sequentially, *Build_CFG* cannot be further divided into multiple parallelizable tasks.

The coarse-grained parallelism between tasks *Build_CFG* and *Build_DFG* allows for the implementation of runtime monitoring on a system of two homogeneous processors, $p1$ and $p2$, each running a separate instance of Pin. $p1$ performs instrumentation and intercepts instructions of basic blocks that match blocks of model B_M. $p2$ waits until $p1$ releases synchronization variable *Lock*, that is, until $p1$ writes intercepted instructions to shared memory. It then reads intercepted instructions and invokes *Build_DFG*, which, using Pin, analyzes the instructions and generates the corresponding data flow. In case the data flow monitored by $p2$ does not satisfy properties embedded within B_M, execution is halted on both $p1$ and $p2$. Otherwise, instructions within each BB_i are allowed to complete execution and commit.

In case a new execution path is detected by $p1$ at a conditional block of B_M, instructions of newly executed basic blocks are intercepted and written into shared memory. $p2$ is responsible for applying the restrictive policies to the newly executed blocks.

Note that memory consistency between $p1$ and $p2$ is guaranteed by setting, resetting, and polling the synchronization variable, *Lock*, in shared memory.

Figure 4 shows the timeline of the runtime monitoring operation implemented on a two-processor system. The x-axis represents the average execution time ratio when runtime monitoring is performed using both software and hardware optimizations. Note that the average execution time ratio is calculated over several real-world

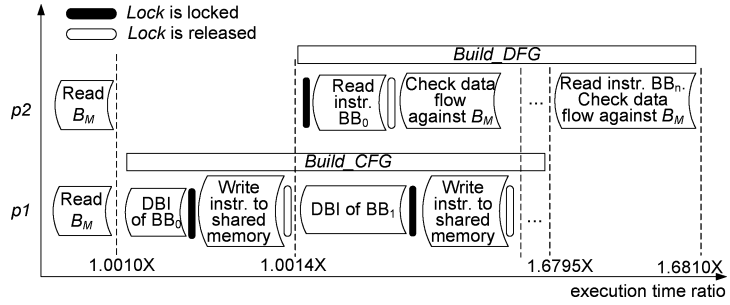

Fig. 4. Timeline of runtime monitoring on a two-processor system.

benchmarks used in the experiments (Section 6). The average execution time in the Real environment, after implementing both software and hardware optimizations, goes up to 1.68X. Thus, we achieve execution time reductions of 27.98% and 37.50%, as compared to the defense framework with proposed software optimizations and the original defense framework, respectively.

6. EXPERIMENTAL RESULTS

We present the results of evaluating the original defense framework and the proposed software and hardware enhancements in this section.

6.1. Experimental Setup

We evaluated the original defense framework and the proposed software optimizations in the context of the QEMU processor emulator (testing environment) and a sharp Zaurus PDA (real environment). The base processor considered in the emulation environment is the ARM926EJ-S processor, a RISC CPU implementing a 32-bit ARM instruction set. The real environment used is a sharp Zaurus SL-5500 [Sharp 2002] running Embedix and Qtopia. It is equipped with a 206-MHz Intel StrongARM processor, with a 32-MB SDRAM, 16-MB Flash ROM, and 128-MB CompactFlash Memory.

The defense framework used in both environments is based on Pin built for the ARM architecture [Hazelwood and Klauser 2006]. Pin is a DBI framework that provides a rich API for building a variety of instrumentation tools. For the purpose of our experiments, we selected applications with known software vulnerabilities (such as bzip2, gzip, ncompress, etc. [Secunia 2007; CERT 2007].) These applications were cross-compiled and patched in order to run on the ARM platform, and were subjected to a large set of input combinations, some of which were specially crafted in order to trigger an exploit of their embedded flaws. We also used several real-world Linux viruses (such as Califax, Dataseg, Kaiowas, etc. [VX Heavens 2007]), cross-compiled for the ARM architecture. In addition, we used our own attack data sets and malicious code, which were only used in the evaluation of the detection effectiveness of the approach. They were not used in the timing analysis of the framework since they do not represent real applications.

We executed the attacks randomly in the testing environment and transmitted extracted B_M's to the Sharp Zaurus, where runtime monitoring in the real environment was performed. Reported execution times in Section 6.2 were based on 40 executions per tested application. Energy measurements were performed using an Agilent 34401A digital multimeter interfaced with a 3.2-GHz Intel Pentium IV PC running Windows. Throughout the execution, we probed the voltage drop across a 0.1-Ω sense resistor, connected in series with the 5-V DC power supply cord to the PDA. The voltage drop across the resistor was sampled at 25 Hz. To calculate the energy consumption, we integrated the power time series using the trapezoidal rule.

Table IV. Execution Time and Energy Consumption in the Real Environment—Original Defense Framework

B.M.	Original defense framework						
	$\#BB_M$	t_1 (s)	t_2 (s)	t_3 (s)	T.R. (X)	E (J)	E.R. (X)
bzip2	1536	42.83	18.18	n/a	2.14	147.07	1.80
gzip	3396	32.96	12.20	n/a	2.94	112.42	2.80
ncompress	2495	32.70	13.10	1.21	2.69	115.02	2.35
rm	3168	2.67	1.26	0.32	2.92	9.93	2.67
sdiff	1296	3.19	1.02	0.30	1.85	10.43	1.72
unzip	5722	28.02	11.57	n/a	3.16	97.35	2.91
zlib	2481	3.60	0.73	0.70	2.61	12.36	2.42
zip	4383	42.20	19.18	n/a	3.03	147.42	2.91
link	3168	1.86	0.87	1.06	2.71	9.17	2.36
gunzip	2581	10.76	4.41	n/a	2.64	36.86	2.39
gcc	4491	4.22	2.16	n/a	3.12	15.84	2.98
tar	3811	25.39	9.30	2.06	2.88	90.24	2.79
touch	4003	2.26	0.97	n/a	2.94	7.88	2.74
Real viruses	2481	3.87	1.62	n/a	2.44	13.51	2.28

In order to test the proposed software optimizations, we used the *objdump* Linux command as the disassembling utility. On the other hand, in order to evaluate the multiprocessor SoC optimizations, we tried multiple multiprocessor simulators (e.g., Simics [2004]; SimIt-ARM [2007], etc.). However, we faced two major problems while building the system: (1) the simulator is unable to simulate a multicore ARM system, or (2) the simulator is not able to load Pin within its memory space. Therefore, we simulated the parallel multiprocessor system by designing a systematic sequential methodology.

In our sequential setting, in order to get the correct functionality of the defense framework, we first executed *Build_CFG* separately on the Sharp Zaurus (acting as processor $p1$) and recorded intercepted instructions into a log file. In order to get a correct approximation of the execution time, we did not account for the time needed to generate the log file. Instead, we took into account the time needed by $p1$ to write into shared memory and set the *Lock* variable, by measuring the time needed by the PDA to write an instruction to shared memory. The log file was then reused by the PDA (now acting as processor $p2$) in order to execute *Build_DFG*. Similarly, for correct execution time approximation, we did not consider the time needed to read the log file. Instead, we considered the time needed by $p2$ to read an instruction from shared memory, and polled and reset the value of *Lock*.

6.2. Results of Software and Hardware Optimizations

This section describes how execution time and energy are affected on the PDA in the testing and real environments.

Execution time in the testing environment is quite significant (goes up to 61.35X). However, this is acceptable and does not impose a severe limitation on the proposed approach, since testing is performed offline (i.e., transparently to the user). Also, since testing is performed in an emulated environment, it does not overwhelm the PDA resources with its security processing time.

In the real environment, performance is evaluated while the runtime monitor is running in parallel with the executed embedded application, and performing checks against model B_M or against restrictive policies. The key elements that constitute the execution time in the real environment are: (1) time required by DBI (t_1), (2) time required to check against model B_M (t_2), and (3) time required for testing a new execution path and rebuilding B_M (t_3). Table IV reports an average of the different time components.

Table V. Execution Time and Energy Consumption in Real Environment—Software and Multiprocessor Optimizations

B.M.	Software optimizations								Multiproc.	
	$\#BB_M$	t_1 (s)	t_2 (s)	t_3 (s)	T.R. (X)	E (J)	E.R. (X)	T.Re./E.Re. (X)/(X)	P.R. (X)	P.Re. (X)
bzip2	1192	35.38	16.79	n/a	1.83	128.39	1.57	1.17/1.15	1.43	1.50
gzip	2614	28.22	9.26	n/a	2.44	92.37	2.30	1.20/1.22	1.67	1.76
ncompress	1796	28.02	10.15	1.21	2.25	96.09	1.96	1.19/1.20	1.66	1.62
rm	2408	2.26	0.98	0.32	2.46	8.40	2.26	1.19/1.18	1.71	1.71
sdiff	975	2.87	0.93	0.30	1.68	9.49	1.56	1.10/1.10	1.38	1.34
unzip	4305	23.38	9.62	n/a	2.63	80.12	2.39	1.20/1.21	2.01	1.57
zlib	1891	3.24	0.63	0.70	2.37	11.08	2.17	1.15/1.11	1.65	1.58
zip	3237	38.20	15.89	n/a	2.67	129.82	2.56	1.13/1.14	1.82	1.66
link	2408	1.66	0.52	1.06	2.32	7.81	2.01	1.17/1.17	1.63	1.66
gunzip	1939	9.49	3.99	n/a	2.35	33.01	2.14	1.12/1.12	1.61	1.64
gcc	3346	3.93	1.61	n/a	2.71	13.48	2.54	1.15/1.17	1.76	1.77
tar	2826	20.33	8.62	2.06	2.43	79.38	2.45	1.18/1.14	1.68	1.71
touch	3022	1.98	0.89	n/a	2.61	7.03	2.44	1.13/1.12	1.95	1.51
Real viruses	1872	3.42	1.39	n/a	2.14	11.79	2.00	1.14/1.14	1.57	1.55

We report execution time and energy measurements for different benchmarks (column 1) in the real environment using both the original defense framework (columns 2–8, Table IV) and the framework implemented with the software optimizations presented in Section 4 (Columns 2–9, Table V). T.R. and E.R. represent, respectively, the execution time and energy ratio as compared to an embedded application running without any instrumentation and checking. T.Re. and E.Re. represent, respectively, the execution time and energy consumption reduction between the original framework and the defense framework implemented with the proposed software optimizations.

In Table V we also report execution time reduction using a multiprocessor SoC (Multiproc.), as approximated using the sequential execution setting explained earlier. Column 10, Table V, reports predicted execution time ratio (P.R.) using the multiprocessor implementation, and column 11 reports the predicted execution time reduction (P.Re.) using both software and hardware optimizations, as compared to the execution time required by the original defense framework.

As can be seen from Table V, the speedup achieved in the real environment, by implementing the proposed software optimizations, was around 1.16X (13.25% execution time reduction). The average energy reduction achieved was 13.15%. The total speedup in executing runtime monitoring on a multiprocessor SoC was around 1.61X (37.50% reduction in execution time) and, consequently, the average execution time on the Sharp Zaurus after deploying the optimized defense framework went up to only 1.68X as opposed to 2.72X when no optimizations were used.

6.3. Detection Results

In this section, we discuss the detection rates achieved by our framework in both the testing and real environments.

6.3.1. Detection Results in the Testing Environment. We have evaluated the effectiveness of the proposed technique in the testing environment using a set of static (Table III) and postexecution security policies, whose specifications are given in Table VI. The system has correctly classified a high percentage of malicious, benign, and exploited vulnerable programs.

Using our framework, we achieved detection rates of 100% for vulnerability exploit detection. We tested our platform on various open-source and synthetic packages that manifest 21 different forms of control and noncontrol data buffer overflow vulnerabilities. We obtained a 100% detection rate, whereas it was reported in Wilander and

Table VI. High-Level Specification of Security Policies

Security policy specification in high-level language
Buffer overflow exploit
$H_1 = \{$ *A return instruction following a buffer overflow and not targeting the instruction after its corresponding call site* $\Rightarrow$ *Security violation*
$H_2 = \{$ *An execve system call following an overflow of allocated memory on the stack/heap/BSS segment* $\Rightarrow$ *Security violation*
$H_3 = \{$ *A call to longjmp following an overflow of allocated memory on stack/heap/BSS segment and its restored environment does not match the one stored at the time setjmp is invoked* $\Rightarrow$ *Security violation*
$H_4 = \{$ *A return instruction following a buffer overflow and not recovering the pushed value of the old base pointer* $\Rightarrow$ *Security violation*
$H_5 = \{$ *An execution path, in which memory overflows beyond the recovered frame pointer* $\Rightarrow$ *Security violation*
$H_6 = \{$ *A call to a function pointer following an illegitimate overwrite of the function pointer* $\Rightarrow$ *Security violation*
$H_7 = \{$ *A GOT entry function call following a buffer overflow and the executed address conflicts with the dynamic relocation section address* $\Rightarrow$ *Security violation*
$H_8 = \{$ *A loop whose condition depends on a variable following a buffer overflow* $\Rightarrow$ *Security violation*
Race condition exploit
$H_9 = \{$ *Inode, symbolic, and hard link discrepancy between* sys_call_1 *and* sys_call_2 *of event-pair* (sys_call_1-sys_call_2) $\Rightarrow$ *Security violation*
$H_{10} = \{$ *Execution path discrepancy between the execution of two paired system calls* $\Rightarrow$ *Security violation*
$H_{11} = \{$ *Inode, symbolic, and hard link discrepancy between subsequent usage of a tmp/ file* $\Rightarrow$ *Security violation*
Malicious code execution
$H_{12} = \{$ *A symbolic or hard link of file X to file Y with root read/write/execute privilege* $\Rightarrow$ *apply* H_{13}
$H_{13} = \{$ *Modifications of file X by instructions within another vulnerable program B and no link operation is contained within B* $\Rightarrow$ *Security violation*
$H_{14} = \{$ *Malicious executable modification* $\Rightarrow$ *Security violation*
$H_{15} = \{$ *Modification of non-executables pointing at instrumented untrusted code or code generated by untrusted programs* $\Rightarrow$ *Security violation*
$H_{16} = \{$ *Non-malicious executable (E) modification* $\Rightarrow$ *Instrument modified executable when executed*
$H_{17} = \{$ *"exec" function variant calls* $\Rightarrow$ *Instrument "exec" argument* (arg_1)
$H_{18} = \{$ *Newly installed programs (P)* $\Rightarrow$ *Instrument new programs*

Kamkar [2003] that the best publicly available buffer overflow detection tool is 50% effective in detecting/preventing the attacks, and there are six attack forms which none of the tools can handle. We obtained a 100% detection rate for open-source and synthetic benchmarks manifesting 11 different forms of race condition vulnerabilities and three different forms of memory vulnerabilities. However, a limitation of our approach is its dependency on the accuracy of the security policies, and the completeness of the set of vulnerability exploits these policies cover.

We also achieved a detection rate of 98.61% for original and obfuscated viruses. Our virus collection consists of 72 real-world Linux viruses [VX Heavens 2007], cross-compiled for the ARM architecture. The virus collection contains both in-the-*wild* (i.e., those currently infecting computer systems) (e.g., the Binom, Bliss, SVAT, and Neox viruses) and in-the-*zoo* viruses (i.e., those that are not being currently spread or are only available online). Furthermore, we obfuscated the available viruses using

Table VII. Comparison of Detection Results Against Antivirus Products

AV product	Detection type	Original viruses (%)	Obfuscated viruses (%)
Linux OS			
Clam	Signature-based	80.55	38.89
F-prot	Signature/heuristics	44.44	1.39
Avira	Signature/heuristics	97.22	50.00
AVG	Signature-based	97.22	34.72
Sophos	Signature/heuristics/ emulation	76.39	0.00
Union AV		**97.22**	**51.39**
Our system		**98.61**	**98.61**

Table VIII. List of Restrictive Policies in the Real Environment

Action	Restrictive policies
Function returns	Not allowed if address value at function call site has not been stored
	If address value is stored, execution is allowed only to the instruction address stored at the corresponding function call site
Frame pointer restoration	Not allowed if corresponding pushed base pointer value has not been stored
	If pointer value has been stored, execution is allowed only if it restores the correct pointer
Function pointer call	Not allowed if following write to allocated memory
GOT entry call	Not allowed if following write to allocated memory
Loop iteration	Not allowed if constraint value follows write to allocated memory
longjmp call	Not allowed if following write to allocated memory
sys_call_2	Not allowed if checks at sys_call_1 have not been performed
	If checks at sys_call_1 have been performed, execution is allowed only if sys_call_2.args comply with sys_call_1.args
tmp_file creation	Allowed to execute only if tmp_file.links $= 0$
"exec" function variants	Not allowed if model B_M was not generated for the executed argument
New installed programs	Not allowed if model B_M was not generated for the programs
Nonexecutable modification	Not allowed if files have root privileges
Executable modification	Not allowed

ELFCrypt [ELFCrypt 2005] and UPX [UPX 2007]. We also ran our tool on multiple benign programs that exhibit behaviors that closely resemble those manifested by computer viruses, such as gcc, javac, gawk, and nasm (45 programs). We also tested our approach on infected versions of these programs.

Our system has correctly classified almost all viruses and benign programs. It falsely declared one Linux virus to be benign. By manually checking the control data flow graph corresponding to the nonidentified virus and its resulting effects on our system, we observed that its execution did not inflict any of the malicious effects specified in our security policies.

Table VII compares the performance of our system, in detecting original and obfuscated viruses, with that of widely used antivirus (AV) products. Detection rates are given in Table VII, columns 3 and 4. We can see that our approach largely outperforms all tested AV tools, and even the combination of all tools (union of the sets of viruses detected by each of the tools), as shown in the **Union AV** row in Table VII.

As mentioned for the case of software vulnerabilities, a limitation of our approach, as can be seen for the case of the nondetected virus instance, is its dependency on the accuracy and completeness of the security policies.

6.3.2. Detection Results in the Real Environment. Behavioral models extracted in the testing environment, in addition to a set of restrictive policies (Table VIII), successfully halted any software exploit in the real environment. For evaluation purposes, we tested the application with (1) user inputs already used in the testing environment, and (2) user inputs not used in the testing environment, potentially triggering new feasible paths. We encountered new execution paths only 8.1% of the time. No false positives

were encountered. However, due to the nature of the utilized restrictive policies, a small percentage of false positives was expected to occur if exercised by adequate inputs.

6.4. Network Traffic

In our proposed framework, the embedded device downloads an untrusted application. If it does not already have the behavioral model, B_M, corresponding to this application, it requests this model from the remote server. Then the server downloads the untrusted application into the simulated embedded environment and tests it for potential exploits. After B_M is extracted, the server uploads it and sends it to the embedded device. Our assumptions are as follows.

(1) The embedded device has Internet access through a wireless or WiFi network or a mobile network (e.g., 3G).
(2) A wireless access point allows the embedded device to connect to the network.
(3) The access point connects to the remote server (a wired device) wired network and relays data between the server and the embedded device.

Based on the preceding, our framework affects network traffic as follows.

(1) The time required for uploading the request of the embedded device for B_M and transmitting it to the access point is negligible, since the request is of an order of few kilobytes.
(2) The time required to relay this request to the wired remote server is negligible as well.
(3) The time required to transmit the behavioral model between the remote server and the access point is also negligible. The incurred delay is calculated as follows: $delay = \frac{Size\ of\ B_M}{Bandwidth\ of\ wired\ network}$, where the size of B_M assumes an average of 10MB and the bandwidth of the wired network could be as high as 2.5 Gb/s.
(4) The only significant delay is the time required to relay the model between the wireless access point and the embedded device. This delay is calculated as follows: $delay = \frac{Size\ of\ B_M}{Bandwidth\ of\ wireless\ or\ mobile\ network}$.

7. RELATED WORK

Various techniques have been proposed for the detection and prevention of software exploits on general-purpose computing systems. Such techniques can be classified into (1) static, (2) dynamic, and (3) hybrid techniques, which combine both static and dynamic analysis. Static analysis techniques examine the source code of a program and detect potential vulnerabilities. FindBugs [FindBugs 2007] is such a tool, which analyzes Java code, looking for security flaws. Due to inherent limitations of static analysis, which typically applies conservative assumptions in order to be tractable, these techniques result in a high false alarm rate. Moreover, accurate static analysis is often not possible when source code is not available.

Dynamic analysis techniques used for software exploit detection rely on real-time execution of a program to test for vulnerabilities. Dynamic analysis has the advantage of observing the application's runtime behavior and its interaction with other system components while it executes. Multiple vulnerability and malware detection tools fall into the category of dynamic analysis [Newsome and Song 2005; Kiriansky et al. 2002; Garfinkel and Rosenblum 2003; Payne et al. 2008; Yin et al. 2007]. Program shepherding [Kiriansky et al. 2002] monitors the control flow of a program and applies a continuum of security rules or policies, ranging from highly restrictive to nonrestrictive rules. These security rules restrict program execution based on code origins and control transfers. Lares [Payne et al. 2008] provides added protection to the security software running on a system that relies on active monitoring of virtualized environment events.

The work shows how to install protected hooks into an untrusted virtual machine. The role of the hooks is to trap execution in the untrusted virtual machine and transfer control to the security application (e.g., antimalware application, antirootkit application, etc.) running on the protected virtual machine. Panorama [Yin et al. 2007] uses taint propagation information at the hardware and operating system levels in order to detect privacy-threatening malware.

A third category of analysis techniques follows a hybrid combination of both static and dynamic analysis. Examples of such techniques can be found in Gupta et al. [1997].

The above-described software exploit detection systems attempt to identify the malicious agent before it starts executing or while it executes. If these solutions identify the malicious agent, they can shut it down and remove it from the system. Since the identification of the malicious agent usually occurs preexecution, most of the above systems are limited to flagging a malicious behavior before it executes, forcing a choice between a pessimistic or an optimistic security rule (i.e., choosing between a high rate of false alarms and a higher probability of undetected attacks). Our approach for software exploit detection and prevention is differentiated in that it allows a program to execute past the point where a malicious behavior is triggered, due to the "safety net" provided by the desktop simulation environment, and execution trace analysis occurs after a program has finished or aborted execution. This is very attractive when a compromised execution closely resembles that of a benign program. Moreover, our approach is based on observing the execution of a program whose source code is not available, modeling safe/unsafe behavior with respect to specified security rules, and ensuring that the program does not deviate from safe behavior. Together, these factors lead to an improvement in detection accuracy.

Moreover, most of the previous work discussed above tackles general-purpose systems and little effort has been directed towards tackling software vulnerability exploits and malicious code execution on embedded systems. Some research efforts have tackled denial-of-service attacks caused by battery exhaustion by designing intrusion detection systems, which take into account the performance, energy, and memory constraints of mobile systems [Nash et al. 2005]. Anomaly detection based on network traffic in mobile networks was presented in Samfat and Molva [1997] and Sun et al. [2004]. In Miettinen et al. [2006], limitations of network-based intrusion detection are discussed and a host-based approach to augment current intrusion detection models for mobile devices is suggested. The host-based approach uses some types of data on embedded systems, such as operating system events. Commercial products [Kaspersky Lab 2007; McAfee 2007] have also been designed to secure embedded mobile devices from viruses and other forms of malware. Our attempt to address host-based software exploit detection on embedded systems is one of the first efforts in this field. We leverage the fact that desktop simulation environments can provide the ability to safely test untrusted embedded applications, without the danger of corrupting the "live" embedded environment or overwhelming it with high computational requirements, in order to design an accurate and optimized defense framework for embedded systems.

8. FUTURE WORK

Our current framework is based on the assumption that the DBI engines and monitoring modules are stealthy and outside the reach of an attacker. As part of our future research, we will leverage the concept of virtual machine introspection [Garfinkel and Rosenblum 2003] in order to keep the DBI engine and monitoring code safe from attacks and reverse engineering. We will also leverage properties of the virtual memory system and page-fault exceptions in order to stealthily instrument vulnerable and malicious code [Vasudevan and Yerraballi 2006]. We will also study whether some functions, which we refer to as *hotspots*, performed by the embedded environment

runtime monitor can be implemented as application-specific custom instructions, thereby accelerating the core operations of those hotspots and further reducing the execution time on the embedded system.

9. CONCLUSION

In this article, we introduced an efficient and scalable approach for the detection and prevention of software vulnerability exploits and malicious code execution on embedded systems. The defense mechanism is based on the use of two isolated execution environments—testing and real environments—DBI, and hybrid software analysis. In the testing environment, our technique successfully detects and thwarts runtime software exploits, through the design of appropriate static and postexecution security policies. The use of the simplified abstracted behavioral model for monitoring execution in the real environment results in an acceptable performance penalty, while providing a high level of security at runtime.

REFERENCES

AARAJ, N., RAGHUNATHAN, A., AND JHA, N. K. 2008. Virtualization-based framework for malware defense. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. 64–87.

CABIR. 2004. Virus descriptions: Cabir. `http://www.disklabs.com/cabir.asp`.

CERT. 2007. Vulnerability notes database. Computer Emergency Response Team. Carnegie Mellon University, Pittsburgh, PA. `http://www.kb.cert.org/vuls`.

CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*. 177–192.

ELFCRYPT. 2005. `http://www.infogreg.com/source-code/public-domain/elfcrypt-v1.0.html`.

FINDBUGS. 2007. `http://findbugs.sourceforge.net`.

FLEXISPY. 2006. Flexispy spills blackberry secrets. `http://www.flexispy.com/news-flexispy-blackberry-windows-mobile.htm`.

GARFINKEL, T. AND ROSENBLUM, M. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium* 191–206.

GUPTA, R., SOFFA, M. L., AND HOWARD, J. 1997. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. Soft. Eng. Meth. 6*, 370–397.

HAZELWOOD, K. AND KLAUSER, A. 2006. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference Software Engineering*. 291–301.

KASPERSKY LAB. 2007. Anti-virus system protects mobile devices. `http://rfdesign.com/next_generation_wireless/news/kaspersky-anti-virus-mobile-devices-0208`.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*. 191–206.

KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. 2004. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*. 18–35.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*. 14–26.

MCAFEE. 2007. McAfee virusscanmobile proven security on the go. `http://us.mcafee.com/root/landingpages/afflandpage.asp?lpname=vs_mobile`.

MIETTINEN, M., HALONEN, P., AND HATONEN, K. 2006. Host-based intrusion detection for advanced mobile devices. In *Proceedings of the Conference on Advanced Information Networking and Applications*. 72–76.

NASH, D. C., MARTIN, T. L., HA, D. S., AND HSIAO, M. S. 2005. Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshop.* 141–145.

NEWSOME, J. AND SONG, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Conference on Network and Distributed System Security Symposium*.

PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*. 233–247.

PERKINS, J. H. AND ERNST, M. D. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*. 23–32.

QEMU. 2008. QEMU: Open source processor emulator. http://fabrice.bellard. free.fr/qemu.

RAVI, S., RAGHUNATHAN, A., KOCHER, P., AND HATTANGADY, S. 2004. Security in embedded systems: Design challenges. *ACM Trans. Embedd. Comput. Syst. 3*, 461–491.

SAMFAT, D. AND MOLVA, R. 1997. IDAMN: An intrusion detection architecture for mobile networks. *IEEE J. Select. Areas Comm. 15*, 1373–1380.

SECUNIA. 2007. Vulnerabilities and virus information. http://secunia.com.

SHARP. 2002. Device profile: Sharp's Zaurus SL-5500 Linux PDA. http://www.linuxdevices.com/articles/AT2134869242.html.

SIMICS. 2004. Virtutech Simics. http://www.virtutech.com/whatissimics.html.

SIMIT-ARM. 2007. http://simit-arm.sourceforge.net.

SUN, B., YU, F., WU, K., AND LEUNG, V. C. M., Oct. 2004. Mobility-based anomaly detection in cellular mobile networks. In *Proceedings of the Workshop on Wireless Security*. 61–69.

UPX. 2007. The Ultimate Packer for eXecutables. http://upx.sourceforge.net.

VASUDEVAN, A. AND YERRABALLI R. 2006. SPiKE: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the Australasian Computer Science Conference* 311–320.

VX HEAVENS. 2007. http://vx.netlux.org.

WEGMAN, M. AND ZADECK, F. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 181–210.

WILANDER, J. AND KAMKAR, M. Feb. 2003. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium.*

YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communication Security*. 116–127.

Received December 2008; revised May 2009; accepted September 2009