On-Chip Control Flow Integrity Check for Real Time Embedded Systems

Fardin Abdi Taghi Abad*, Joel Van Der Woude*, Yi Lu*, Stanley Bak*
Marco Caccamo*, Lui Sha* , Renato Mancuso*, Sibin Mohan†

*Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801
Email: {abditag2, jvande31, yilu1, sbak2, mcaccamo, lrs, rmancus2, sibin}@ILLINOIS.EDU

***Abstract*—Modern industrial plants, vehicles and other cyber-physical systems are increasingly being built as an aggregation of embedded platforms. Together with the soaring number of such systems and the current trends of increased connectivity, new security concerns are emerging. Classic approaches to security are not often suitable for embedded platforms.**

In this paper we propose a hardware based approach for checking the integrity of code flow of real-time tasks whit precisely predictable overheads that do not affect the critical path. Specifically, we employ a hardware module to perform control flow graph (CFG) validation at run-time of real-time component. For this purpose, we developed a binary-based, CFG generation tool. In addition, we also present our implementation of a CFG integrity checking module. The proposed approach is aimed at improving real-time systems security.

I. Introduction

Many safety-critical systems such as advanced automotive/avionics systems, medical equipment, transportation, power plants and industrial automation systems employ embedded real-time components. These appear in the roles of controllers, physical sensors or other tasks, depending on the actual system. These components are usually responsible for the most critical tasks in the system. In most cases, the proper functioning of such components is of the utmost importance; otherwise, they could lead to loss of life and/or injury to human beings and also result in significant damage to the system(s) and/or environment.

In the past embedded systems were less vulnerable to malicious activities when they were not connected to the network [34]. However, recently such systems are more interconnected or even controlled over the internet. Moreover, there is more monetary and adversarial motivations for malicious activities in recent times. Examples of such malicious activities include: the W32.Stuxnet worm that highlighted the possibility and effectiveness of an attack on a nations critical infrastructure [29], malicious code injection into the telematics units of modern automobiles [7], [16] and attacks on UAVs [25] among others. Such attacks motivate a closer inspection of the security of industrial control systems everywhere.

The material presented in this paper is based upon work supported by National Science Foundation (NSF) under grant numbers CNS-1035736 and CNS-1219064 and the Department of Energy under Award Number DEOE0000097. This report was prepared as an account of work sponsored by an agency of the United States Government (e.g. NSF, DOE). Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

While general-purpose processors are opening doors for new defensive techniques through increased parallelism, clock speed and memory size, many real-time embedded systems are constrained by timing requirements and small onboard memory sizes. Moreover, schedulabiliy of hard real-time tasks needs to be guaranteed, even with the additional overheads incurred by (any) security techniques. Given their tight constraints, traditional techniques to prevent against cyber attacks are not necessarily feasible; Many software based security mechanisms developed for general purpose systems create large workloads that are difficult to schedule. Moreover, they either require components that do not necessary exist in simple embedded system (such as trusted operating system or memory management units) or the overheads imposed by them is not predictable enough for providing guarantees that are necessary for such systems. New defenses that are designed with the limitations of embedded/realtime systems in mind are required to adequately protect these systems.

In this paper we present *a hardware based security approach with predictable overhead* for embedded real-time systems. We check for the *control flow integrity of the real-time task*. We propose adding an on-chip control flow monitoring module (OCFMM) to the processor core with its own isolated memory unit. OCFMM has direct hooks into the processor that enable it to track the control flow of the program. we also present methods for *(a)* determining the control flow graph (CFG) of tasks that execute on the processor and *(b)* loading them on to the memory units in advance. OCFMM monitors the run-time control flow and compares it to the stored CFG. Hardware implementation of OCFMM and the isolation of its memory unit from the rest of system eliminates the possibility of any attack on the OCFMM unit itself. We also take advantage of a hardware stack mechanism to keep track of call and return addresses (similar to SmashGuard [21]). In this paper, we also present our tool for generating the control flow graph from a binary without the need to modify the binary or access to source code.

Our proposed technique effective and applicable to embedded real-time systems for three main reasons. First, software updates of these types of systems is rare. In fact, their software is updated only when the system administrator needs to change the application of the system or when some system settings are modified. Therefore, any overheads due to the generation a new CFG profile of the program and loading it into the OCFMM is acceptable. Second, the overhead incurred by OCFMM is finite and predictable. Therefore, it is well-suited for hard real-time systems in need of formal ver-

978-1-4799-0798-4/13/$31.00 ©2013 IEEE

ification. Finally, our technique does not require components that may not exist in simple embedded systems (*e.g.,* trusted operating systems or memory management units). The security guarantees and predictability provided by our OFCMM technique, make this technique very effective for embedded real-time components in safety-critical systems. .

In the following sections first, we introduce different types of attacks, then describe the attack model and our assumptions. In section III-C, the approach for generating control flow graphs and also OCFMM design and its detection algorithm is described. In section IV, some experiments in addition to performance analysis is presented. Section VI talks about related work and the last section concludes the paper.

II. Attacks

In this section we provide an overview on the popular types of attacks on embedded systems that modify control flow of the program. Monitoring the control flow of the program would help us detect these attacks.

A. Buffer Overflow

The CWE/SANS list of the top 25 most dangerous programming errors lists buffer overflows as the third most dangerous vulnerability [3]. A buffer overflow is an attack that exploits a lack of sanitization in performing I/O operations which involve inbound data transfers. Specifically, when the bounds of incoming data are not checked properly, an attacker is able to write more data than intended, overwriting nearby data and compromising the code logic [20]. Buffer overflow vulnerabilities can affect different regions of a program's memory, such as stack or heap.

Exploiting a stack overflow, the attacker is able to overwrite the return address of the current stack frame. In this way, it is possible to redirect the execution to an almost arbitrary code block. The other common case of a buffer overflow is called heap overflow. In a way that is similar to what explained above, heap overflow vulnerabilities allow an attacker to write data on the heap exceeding the boundaries of a buffer. The same techniques employed in a stack overflow attack can be set in place.

It should be noted that in both stack overflow and heap overflow attacks, another way to redirect the execution flow is to overwrite a control variable, such as a branching condition. In this way, the attacker is able to alter the normal behavior of the code flow, for instance driving the execution into unauthorized sequences of instructions.

B. Return-into-libc

Return-into-libc is an attack which leverages on a buffer overflow in order to direct execution towards a `libc` function that has been included in the compiled binary, thereby requiring no shellcode to be injected. For instance, a call to the library function `system()` can be forged, passing /bin/sh as an argument [2] and leading to the execution of a command line interpreter. Return-into-libc is a particularly powerful attack because it is able to bypass protection measures that disable the execution of code from stack or heap regions: one of the most common defenses against buffer overflows attacks [1].

C. Return-oriented-programming

Like the return-to-libc, return-oriented-programming is a more general technique that allows the execution of malicious code bypassing memory protection defenses. As explained in [23], once an attacker is able to overwrite a function return address, he can chain the execution of small preexisting code fragments to produce arbitrary program behavior. A known technique involves searching for instructions that alter the control flow, typically return instruction, and then scanning preceding bytes for instructions that can be used for the attack. Such instruction snippets are called "gadgets" and are chained together via a buffer overflow exploit. Specifically, the vulnerable stack is injected with the sequence of addresses of the employed gadgets, making sure that the address of the first gadget overwrites the original return address. Once the first gadget is executed, the trailing return instruction determines the next on the stack to be executed, thus giving control to the second gadget. Similarly, subsequent gadgets are executed and the desired behavior is produced. According to [23], any sufficiently large quantity of code can contain a set of gadgets that are Turing-complete, providing full functionality.

D. Code injection

The term "code injection" refers to a technique that leverages on the control over an existing process in a system (e.g. an already compromised one) to spread the attack to other software components of the same system. Usually, the aim is to inject code into a process with high privileges from a low-privileged one. As explained in [35] this type of attack can be carried on using APIs of the `NTCreateThread()` family on Windows platforms. In this way, the attacker is able to execute any function which exists in the context of the targeted executable. On Linux platforms, the `ptrace()` system call can be used to manipulate the execution flow of the attached process. Additionally, on both Windows and Linux platforms, code injection can be performed leveraging on APIs that allow programs to load shared objects into running processes. DLL (SO respectively) injection attacks can be performed against Windows (Linux respectively) platforms in order to gain full control of open file descriptors, intercepting I/O and executing functions within the context of the victim process [35].

III. Control Flow Monitoring

In this section we detail the proposed methodology. The assumptions under which we are able to perform threat detection are presented. Furthermore, a detailed explanation of how CFG run-time checking can be effective against the attack techniques presented in the previous section is provided.

A. Trusted Initialization

The proposed technique requires the system to be initialized safely. As explained in section III-C, together with the standard system bootstrap procedure, a CFG needs to be correctly produced from the binary and loaded into the OCFMM memory.

It is important to underline that the assumption of a trusted initialization sequence is realistic for embedded systems. Firstly because the supply chain for industrial controllers and systems is generally supervised. Second, because it is often the case that before such systems become operative, physical access to them is restricted to trusted personnel only. Third, because typically network connections are established only after the bootstrap sequence is completed. Once the initialization sequence completes and the proposed detection module is booted, the system can establish connections to untrusted networks and become fully operational. Since a trusted initialization is performed at system boot, there is no need to secure bootstrap code executed before the critical

components are loaded and the detection module starts. Instead, intrusion detection and execution flow inspection can be performed after the initialization phase is completed.

B. Attack Model

As previously mentioned, we do not consider attacks which require physical access to the targeted system. The reason is twofold. First, because our focus is on embedded devices, with particular attention to industrial plant control systems, whose physical access is generally restricted. Moreover, assuming untrusted physical access to this class of systems raises a whole gambit of options to maliciously impact the functionality.

On the other hand, we do assume that the attacker has networked access to the system, by means that an interface is available through a remote connection or indirect channels. For example, it includes attacks that are set in place infecting USB devices that are connected at a later stage to the targeted system or to an intermediate machine that is networked with the fianal target. We also assume that the industrial control system has zero-day vulnerabilities, unknown to the administration personnel, which could be leveraged by an attacker. This is a realistic assumption given the lifecycle of many embedded devices, as they can be running legacy code for decades without being updated or rewritten. Furthermore, we are following an open design policy, by means that we assume an attacker to have access to the source code of the program on the controller and a complete knowledge of the hardware design.

In this paper we are only targeting the attacks that are altering the control flow graph. However, our technique does not offer sufficient protection from attacks that modify the data values of the victim process without diverting its behavior from a legal sequence of execution blocks. These attacks can harm system in two main ways. First, attackers can overwrite the value of function pointers, redirecting execution to an unauthorized code fragment. In order to address this issue, it is enough to prevent the executable code from using pointer-based function calls. This is a reasonable assumption for embedded real-time controllers, whose code exhibits a high level of determinism. Second, attackers can manipulate the values that affect the physical behavior of the considered system. For instance, tampering the variable encoding the target temperature in a thermal controller, would impact the safety of the physical plant. In order to prevent such attacks, Simplex [22] can be used as an additional safety envelope for the output channels.

C. Architecture

The majority of the attacks discussed in Section II share a fundamental aim: arbitrary code execution. Thereby, it follows that by monitoring the execution flow of a program, and cross-validating it against the expected CFG, we are able to detect attacks that maliciously affect the execution flow. We refer back to [5] for the formal analysis of this method.

Figure 1 presents the high-level architecture. The OCFMM is placed on the same chip with the processor and can directly access program counter (PC) and instruction register (IR) with the provided hooks into the processor. Accessing PC and IR does not require any additional operation to be executed on the processor.

Generating the control flow graph of the executable program we want to protect is the first step of our technique. We first describe the control flow graph generation procedure and then detail the OCFMM.

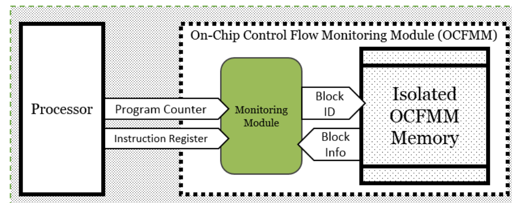

Fig. 1. High level architecture of a processor with OCFMM

D. Control Flow Graph Extraction

We have developed a tool to generate the CFG from compiled binary. This approach guarantees backward compatibility with legacy code bases, without the need to rewrite existing executables or having access to their source code. It is also flexible across different architectures with only minor changes required in order to adapt the CFG generation procedure to the targeted ISA.

We define a "block" as the basic unit of a control flow. Each block is defined as the longest sequence of instructions contained between two control flow statements, i.e. between two statements encoding any variation of a branch, jump, call, or return instruction. Each block is described using three pieces of information: a unique block ID; the address of the first instruction in the block; and its size (number of instructions in the block).

First, during an initialization step, we scan the whole binary executable to identify blocks, according to the mentioned criteria, and to assign unique IDs. Next, starting from the block which contains the entry point of the executable under analysis, it is possible to incrementally build the control flow graph. Specifically, for the block considered at each iteration, we need to know which blocks can be reached next. For conditional jump/branch instructions, we identify a Yes-Block and a No-Block, as the blocks which execute after the current block if the condition of the jump/branch is satisfied or not, respectively. It is enough to store the IDs of said destination blocks in the final CFG. In general, Yes-Blocks can be directly extracted from the branching instruction itself, while No-Blocks start with the subsequent instruction, or vice versa[1]. For unconditional jumps and branches, as well as direct function calls, there is no distinction between Yes-Blocks and No-Blocks, so that just the ID of the target block is stored.

For return instructions, there is no need to store any Yes/No-Block because the execution return address depends on the memory offset of the corresponding call instruction. We developed a hardware stack in order to handle call/return instructions as explained in the next subsection.

The control flow of a program can be seen as a graph $G = (V, E)$, where each vertex $v \in V$ represents a block and each edge $e \in E$ represents a valid control flow between two blocks. An example of control flow graph for the code shown in Figure 2a is reported in Figure 2b.

Despite the existence of loops in the graph, we can use a graph traversal algorithms to get the control flow information of each block. In this paper, we use Depth First Search (DFS), starting at the `main` function. Therefore, this will be the root of the spanning tree, and will be called "main" block.

As previously mentioned, the first step to generate the CFG is to

[1]The arrangement of blocks after branching instructions depends on the compiler used to generate the binary file. Branch prediction strategies may vary how a conditional jump is encoded from the source code.

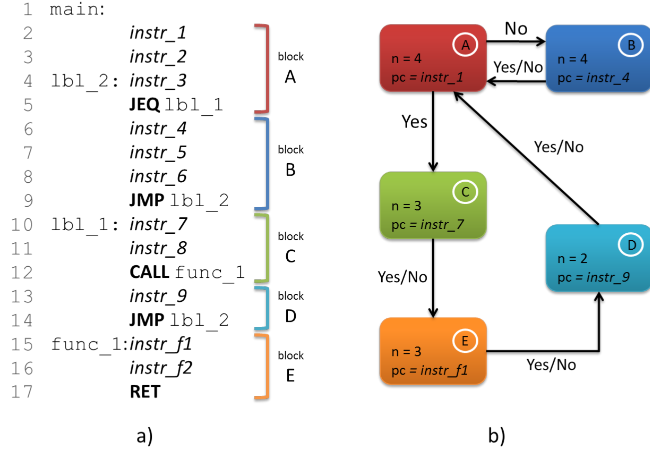

Fig. 2. Assembly code of a sample program (a) and resulting control flow graph (b).

parse the disassembled code into distinct blocks and to assign each block a unique ID. Once this step is done, our algorithm proceeds from the main block. The algorithm which generates the complete control flow graph in a recursive manner is described in figure III-E. The main block has ID = 1, so that the CFG generation begins by calling `recursive_CFG(1)`.

Algorithm 1 recursive_CFG(current_block)

```
if current_block.processed == true then
    return
else if current_block.last_inst is unconditional jump or branch or direct call then
    current_block.processed = true
    current_block.Yes-Block = target_block;
    current_block.No-Block = target_block;
    recursive_CFG(target_block);
else if current_block.last_inst is Conditional jump or branch then
    current_block.processed = true
    current_block.Yes-Block = target_Block ;
    current_block.No-Block = current_block + 1 ;
    recursive_CFG(current_block.Yes-Block);
    recursive_CFG(current_block.No-Block) ;
else if current_block.last_inst is return then
    current_block.processed = true
    current_block.Yes-Block = 0 ;
    current_block.No-Block = 0 ;
end if
return
```

Finally each block profile contains starting address of the block, IDs of Yes-block and No-Block and block size. Conversely, For indirect calls or jumps (as they can result from a function pointer or a jump table) we must enumerate all the possible instruction targets. This enumeration requires a more in-depth analysis and it is currently left as a future work. A naïve implementation would be allowing an indirect call or jump to any target memory address. However, this would determine security issues, e.g. exposing to return-to-libc attacks.

E. Detection Mechanism

At a high-level description, the detection module is in charge of comparing the control flow of the running program with the CFG loaded into the dedicated memory at boot time. If a mismatch occurs, it raises a detection flag.

In the proposed design shown in Figure 1, the OCFMM storage unit is where the CFG profile of the program is loaded. When execution begins, the profile for the main block (with ID equal to 1) is fetched. Thus, what are the next valid states becomes available to the detection module.

Similarly, during the execution of a given block, CFG profiles for the possible next blocks are pre-fetched. Our module continuously checks that the execution remains inside the current block. On the other hand, whenever it detects that a change in the executing block occurs, it validates the current address of the program counter against the possible, previously fetched destination addresses. If there is a match, execution is not affected, otherwise a detection flag is raised, allowing the system itself or the administration personnel to take appropriate actions.

As previously stated, in addition to conditional and unconditional branches, it is necessary to handle function calls. In order to correctly handle call/return instructions we rely on a stack module implemented in hardware using a FILO buffer. When a function is called, the address at which to return is pushed onto a hardware stack. When the function call returns, the address of the corresponding call is popped from the top of the stack. The detection module verifies that execution is resumed at such address. Again, if a mismatch is observed, detection flag is raised. The main idea behind this implementation is similar to SmashGuard [21].

The basic algorithm used to check control flow integrity is reported below. This algorithm is executed every time that the value of PC changes. In this algorithm, PC is the current value of the program counter and B is the descriptor of a block of control flow, which contains $\{pc, n, Yes\text{-}Block, No\text{-}Block\}$. $B.pc$ represents the address at which the control flow block B begins. $B.n$ is the number of instructions contained in block B. Finally, $B.Yes-Block$ and $B.No-Block$ encode the control flow blocks that may occur after B, as explained in Section III-D.

F. Predictable Over Head

In hard real-time system, safety and reliability of system is only guaranteed if critical tasks of the system finish before their relative deadlines. Therefore, system designers need to perform extensive schedulability analysis to ensure this condition would not be violated. Due to this, techniques that alter execution time of the tasks in an unpredictable manner, cannot be applied to this class of systems such as most of the techniques employing randomness or obfuscation.

Having this constraint in mind, we have designed our technique such that incurred overhead on each block is precisely predictable. The reason of the overhead is that for some very small execution blocks, the time required to load the block information from OCFMM memory is longer than the time required for executing the instructions of that block, therefore execution needs to be halted until the block information is fully loaded.

However, for each block, the upper bound on the halting time could be simply calculated. If e_i is the minimum execution time of ith instruction in the block, m is the access time for OCFMM memory and n_k is the number of instructions in kth block. Then we have the following for upper bound of overhead on kth block,

Algorithm 2 Detection Algorithm

```
if Instruction is a call then
    push PC + 4
end if
if Instruction is a return then
    newB = pop()
    if newB.pc == PC then
        B = newB;
    else
        raise detection flag;
    end if
else if PC == (PC_Previous + 4) AND (PC < B.pc + B.n) then
    return ;
else
    if PC belongs to B.Yes-Block then
        return
    else if PC belongs to B.No-Block then
        return
    else
        Raise detection Flag
    end if
end if
return
```

$$\text{overhead}(k) = \begin{cases} m - \sum_{i=1}^{n_k} e_i, & \text{if } m > \sum_{i=1}^{n_k} e_i \\ 0, & \text{otherwise} \end{cases}$$

With the worst case overheads of every block, upper bound of overhead on every piece of program is calculable, too. System designer can use this information to generate new worst case execution times for tasks in order to verify schedulability of the task sets in hard real-time systems.

IV. Experiments and Evaluations

In order to test and verify the applicability of our approach, we integrated our proposed control flow integrity checking module into the design of the LEON3 soft-core processor. LEON3 is a 32-bit synthesizable VHDL model of a SPARC V8 processor. We implemented a single core processor without virtual memory on a Xilinx Virtex-5 LXT FPGA ML505 evaluation platform. We utilized excess fabric on the FPGA to implement our module with hooks into the 7-stage pipeline of the LEON3. Our aim was to introduce minimal changes to the design of the LEON3 while being able to monitor CFG by only hooking into the program counter (PC) and instruction register (IR).

In our prototype implementation due to limitations of the FPGA board, we did not add an external memory unit for OCFMM. Instead we implemented a SRAM unit on FPGA fabric. This implementation works for our prototype experimental set up while it restricts maximum size of the CFG profile and consequently the size of our program. For future work, we will implement an external storage memory unit for OCFMM. Having a dedicated memory unit eliminates any limitations on the size of executable in addition to loading CFG profile during update. Due to limited size of hardware stack, we ran our preliminary experiments under the assumption that there is no nested calls deeper than 200 levels.

We performed the first experiment on a PID controller designed for temperature control in an industrial unit. The program reads a value entered by user as a reference temperature and generates a relative output signal. PID controller code consists of the following components: simple function for reading the reference input and the control loop. In every execution of the loop, first the function responsible for reading the sensor values is called. Next, output is calculated based on the read values and propagated on the output port. After running the CFG generation tool, 240 execution blocks were detected. In this version of our implementation, size of information of each block is 9 bytes. Consequently, size of CFG profile generated for this program 2160 bytes.

In the first experiment, we simulated the code replacement attack by loading a modified binary onto the processor where one of the jump destinations is different from the expected address resulting in a different CFG. OCFMM was able to detect the mismatch and detection flag was raised.

In the second experiment, we simulate control flow graph modifications due to overwriting a return address in stack. We used the same program as the previous experiment where the value read from sensor is written into an unbounded buffer. A malicious sensor output written to this buffer would overwrite the return address in the stack, and would redirect the execution flow to the address of attacker's desire. Our approach successfully detected the CFG mismatch with the expected CFG and raised the detection flag.

V. Future Work and Limitations

There is much more work to be done to improve our technique and expand its effectiveness and applicability.

Our future plan is to further advance our implementation by replacing on-chip SRAM unit of OCFMM with an external one. Additionally, in order to mitigate the overhead caused by halting execution of small blocks or slow external memories, we are planning to use a CFG profile caching mechanism to pre-fetch profile of multiple levels in advance. This would eliminate the need to halt the processor for short blocks as the information are already present in the cache, consequently reducing the overhead.

Even though our technique, as proposed in this paper, provides a predictable overhead, but since the implementation is still incomplete, we have not yet provided comprehensive measurement on the overhead.In our future work, along with a caching mechanism and an external memory, we plan to perform extensive measurements on performance and logic overhead

We hypothesize that by using additional hooks into the processor to determine current privilege level stored in the MMU, our module would be able to distinguish between multiple tasks and monitor the control flow of each. Securing the whole system by detecting and securing some critical components of the system is the another direction of research that we are currently working on. Finally, additional research into methods of preventing state variables from being overwritten by a buffer overflow is needed to provide reasonable protection against that form of attack.

VI. Related Work

There has been a large body of work on mitigating control hijacking attacks. due to lack of space, we focus on most relevant and well-known works.

There is a line of research that performs software based control flow monitoring. In [30], the CFG of a program is extracted from static analysis of program source code and used to build the models for system calls. Program shepherding [14] is another technique that performs function level control flow monitoring. It extract CFG during the execution of the program. [4] also monitors the

program in function level but it extracts the CFG from binary. The problems of all these software based approaches is the large overhead ranging from 45% in control-flow integrity to 660% in program shepherding.

Moreover, there is another direction of research that uses secrets to protect the control flow integrity. Point guard [9], stores code addresses in an encrypted form in data memory. [6] uses address obfuscation for security purposes. [8], [11], [28], [33] also rely on secret values in order to prevent intruder from being able to easily predict and modify pointer addresses. Exploiting these techniques opens an additional vulnerability (keeping the secret values). Some of these techniques have also been found to be vulnerable [24], [27].

Additionally, there are number of other software/hardware defenses against control flow altering through buffer overflow. [10], [12]–[14], [18], [21] have limited protection compared to our approach. For instance some of them fail to protect from return-to-libc or only cover attacks on function return address.

Moreover, directly trying to integrate security solutions that have been developed for general purpose systems to embedded RTS [13], [15], [17], [19], [26], [31], [32] may not always be the best solution. These are not always aware of the underlying nature of embedded RTS, *e.g.,* attempts to integrate cryptography into embedded RTS should be compliant with the strict scheduling policies of such systems or some of these techniques require trusted operating system.

VII. Conclusion

In conclusion, we have introduced on-chip control flow monitoring to enforce control flow integrity in embedded real-time systems. The solution presented is well suited for simple embedded real-time systems that do not have a Memory Management Unit (MMU) and have strict timing requirements. Additionally, it introduces finite and predictable overhead in terms of execution time and does not require any modifications to the source code or any sort of binary rewriting. This allows the technique to be applied to legacy code, propriety programs, or third party products. This technique was able to detect modifications to the control flow graph of an executing program preventing the attackers from return-into-libc, return-oriented-programming, and code injection. The technique was successfully implemented as a modification of LEON3 soft core.

References

[1] Getting around non-executable stack @ONLINE, 1997.
[2] Advanced return-into-lib(c) exploits (PaX case study) @ONLINE, 2001.
[3] 2011 cwe/sans top 25 most dangerous software errors @ONLINE, 2011.
[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.
[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.
[6] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium*, volume 120. Washington, DC., 2003.
[7] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, Aug 2011.
[8] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, volume 3, 2001.
[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.
[10] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998.
[11] R. Doe. The pax project @ONLINE, 2004.
[12] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26. ACM, 2009.
[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.
[14] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX security symposium*, volume 6, 2002.
[15] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi. Security as a new dimension in embedded system design, 2004.
[16] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447 –462, may 2010.
[17] M. Lin, L. Xu, L. Yang, X. Qin, N. Zheng, Z. Wu, and M. Qiu. Static security optimization for real-time systems. *IEEE Transactions on Industrial Informatics*, 5(1), Feb. 2009.
[18] M. Milenković, A. Milenković, and E. Jovanov. Hardware support for code integrity in embedded processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 55–65. ACM, 2005.
[19] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM Conference on High Confidence Networked Systems*, 2013.
[20] A. One. Smashing the stack for fun and profit @ONLINE.
[21] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. *Computers, IEEE Transactions on*, 55(10):1271–1285, 2006.
[22] L. Sha. Using simplicity to control complexity. *Software, IEEE*, 18(4):20 –28, jul/aug 2001.
[23] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.
[24] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.
[25] D. Shepard, J. Bhatti, and T. Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle. *GPS World*, August 2012.
[26] S. Son, R. Mukkamala, and R. David. Integrating security and real-time requirements using covert channel capacity. *Knowledge and Data Engineering, IEEE Transactions on*, 12(6):865 –879, nov/dec 2000.
[27] A. N. Sovarel, D. Evans, and N. Paul. Wheres the feeb? the effectiveness of instruction set randomization. In *14th USENIX Security Symposium*, volume 6, 2005.
[28] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 209–220. IEEE, 2004.
[29] US-CERT. ICSA-10-272-01: Primary stuxnet indicators. Aug. 2010.
[30] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168. IEEE, 2001.
[31] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.
[32] T. Xie and X. Qin. Improving security for periodic tasks in embedded systems through scheduling. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.
[33] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 260–269. IEEE, 2003.
[34] M.-K. Yoon, S. Mohan, and L. Sha. Securecore: A multicore architecture for intrusion detection in real-time control systems. In *IEEE Conference on Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2013.
[35] G. Yucheng, W. Peng, L. Juwei, and G. Qingping. A way to detect computer trojan based on DLL preemptive injection. In *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2011 Tenth International Symposium on*, pages 255–258, 2011.