Defending Embedded Systems Against Control Flow Attacks

Aurélien Francillon
INRIA Rhône-Alpes
francill@inrialpes.fr

Daniele Perito
INRIA Rhône-Alpes
perito@inrialpes.fr

Claude Castelluccia
INRIA Rhône-Alpes
ccastel@inrialpes.fr

ABSTRACT

This paper presents a control flow enforcement technique based on an Instruction Based Memory Access Control (IBMAC) implemented in hardware. It is specifically designed to protect low-cost embedded systems against malicious manipulation of their control flow as well as preventing accidental stack overflows. This is achieved by using a simple hardware modification to divide the stack in a data and a control flow stack (or return stack). Moreover access to the control flow stack is restricted only to return and call instructions, which prevents control flow manipulation. Previous solutions tackled the problem of control flow injection on general purpose computing devices and are rarely applicable to the simpler low-cost embedded devices, that lack for example of a Memory Management Unit (MMU) or execution rings. Our approach is binary compatible with legacy applications and only requires minimal changes to the tool-chain. Additionally, it does not increase memory usage, allows an optimal usage of stack memory and prevents accidental stack corruption at run-time. We have implemented and tested IBMAC on the AVR micro-controller using both a simulator and an implementation of the modified core on a FPGA. The implementation on reconfigurable hardware showed a small resulting overhead in terms of number of gates, and therefore a low overhead of expected production costs.

Categories and Subject Descriptors

K.6.5 [**Operating Systems**]: Security and Protection

General Terms

Experimentation,Security

Keywords

Control flow attacks, Stack-based buffer overflow, Software security, Return stack, Return-oriented programming

1. INTRODUCTION

Embedded systems are commonly used for safety critical applications and can be deployed in hostile environments. In applications like industrial control systems, automotive systems, both civil and military applications a correct operation of the embedded devices might be indispensable. However, as the connectivity of these devices with the outside world increases, the risk of being be remotely subverted increases as well.

Computer systems are subject to remote attacks that aim at controlling their software behavior, which often require control flow manipulation. Such attacks, that we refer to as *Control Flow Attacks*, have been one of the main attack vectors to computer systems in recent years. Embedded systems are not an exception to this and, despite their limited computation capabilities, several attacks have been recently shown to be practical and feasible on them [11, 12].

Given the high impact that control flow attacks had on commodity systems, many countermeasure techniques have been proposed to defend against such attacks, such as: binary randomisation [14], memory layout randomisation [20, 21], stack canaries [9], tainting of suspect data [19] enforcing pages to be writable or executable [3, 21], Control Flow Integrity enforcement [1]. However, most of those countermeasures are demanding in terms of computation capabilities, memory usage and often rely on hardware that is unavailable to simple micro-controllers such as a Memory Management Unit (MMU) or execution rings. Moreover, they mostly use software solutions as hardware modifications (for example on the IA-32 architecture) are difficult and likely to cause problems with legacy applications.

In this paper we introduce a simple but effective hardware protection against control flow attacks that we implemented on the AVR family of micro-controllers, a very common architecture in wireless sensor networks and in low-end embedded systems. The defense relies on using a separate stack for storing return addresses. This *Return Stack* is stored in data memory at a different location than the normal stack and is protected in hardware against accidental or malicious modification.

The technique has been implemented and validated on both a simulator and an AVR core on a FPGA (i.e. a soft-core). The prototype has been implemented on the AVRORA [22] software simulator and in VHDL.

This demonstrates the possibility to implement this feature with a modest overhead in terms of logical elements units, with no run-time impact, and backward compatibility on all major software functionality. In order to support

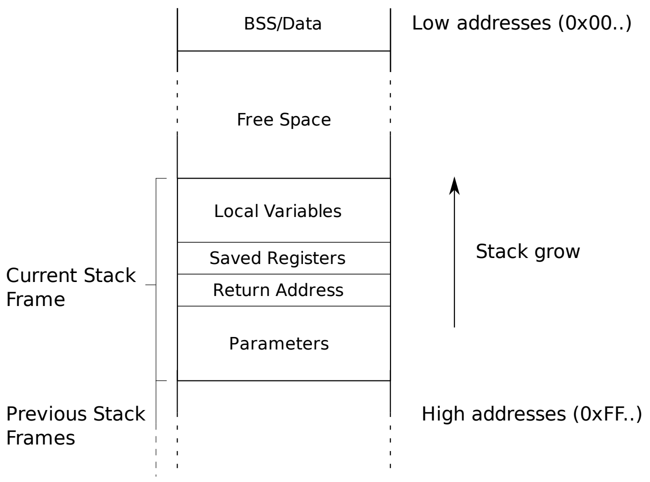


Figure 1: Normal function frame layout after a function call.

this feature the device needs application specific configuration to be performed at boot time. This configuration is performed during the very first step of software initialization and therefore can be performed by the C library after basic initialization of memory. Apart from this change the compiler libraries and programs do not need modifications.

Besides defending against attacks this stack layout can also be very helpful for software reliability to prevent stack overflow.

2. PROBLEM STATEMENT

2.1 Control flow attacks

During the execution of a call instruction, the processor transfers control to the code that implements that procedure. When the procedure completes, control is transferred back to the instruction following the call instruction. On most microprocessors a unique stack is used to store control flow information as well as other data. Each *frame* of the stack usually contains the following data:

- saved return addresses of the caller;
- function variables and parameters;
- saved registers, according to the specific Application Binary Interface (ABI).

Implementation details vary across different architectures, but in Figure 1 we depict a common layout for a portion of the stack. As it is possible to note, control flow information, like return addresses, are stored alongside other function data.

When the data stored in the function local variables comes from untrustworthy sources and sufficient checks are not in place, memory corruption of the local variables might occur. Corruption that can allow the attacker not only to taint the function data but also corrupt the control flow information stored on the stack.

2.2 Stack overflow

Stack overflow is an out of memory condition common in embedded systems with highly constrained memory availability. This is the definition we will use throughout this paper and it must not be confused with *stack based buffer overflow*. The latter is the consequence of a buggy program (e.g. improper boundary check) and the former is the consequence of an out-of-memory condition and can occur with a correct program or a program written in a type (or memory) safe language.

Stack overflows are common on simple micro-controllers, due to their limited memory size. This condition can occur, for example, when too much data is allocated on the stack or when the depth of the stack grows too large. In both cases, the stack exhausts its available memory and overlaps with other memory sections like the BSS section.

This is both a reliability problem and a security problem. It is a reliability problem as the stack overflows in other memory regions, it can corrupt the data stored there. This usually leads to bugs that are difficult to track. Because, for example, the corrupted variable will depend on the layout of variables in the BSS section, this depends on how the compiler will order variables in memory. It can be a security problem as an attacker might take advantage of a stack overflow to overwrite a return address without any specific program vulnerability. When this function will return the control flow will be directed to the address chosen by the attacker.

Stack overflow conditions are easily detected in general purpose operating systems where a page fault occurs when memory is accessed beyond the currently allocated stack space. When this page fault occurs the operating system can take appropriate actions. However, this solution is not feasible when an MMU is not available.

For embedded systems the stack consumption can be analyzed before execution performing static analysis on the program [16]. Static analysis will reveal whether the device will have enough memory to execute the application. However in some cases it can be difficult to know exactly the maximum stack consumption, for example:

- when indirect calls are present the tool has to perform data flow analysis, which is not always feasible,
- when re-entrant interrupts are used the call depth could be unbounded,
- if recursive function calls are performed, data flow analysis would have to be performed, if possible.
- some compilers implement a way to allocate dynamic memory on the stack as non standard extensions, for example *gcc* provides the alloca built-in function for this purpose. This is again a difficult case for static analysis tools.

3. INSTRUCTION BASED MEMORY ACCESS CONTROL FOR CONTROL FLOW INTEGRITY

3.1 Overview of our solution

The main idea behind IBMAC is to protect return addresses on the stack from being overwritten with arbitrary data. By doing so, as we will show later, IBMAC also protects embedded systems from memory corruption caused by stack overflows.

The intuition is that control flow data should be only read and written by the `call` and `ret` family of instructions

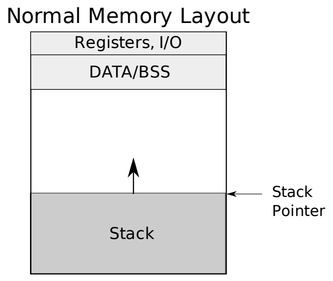


Figure 2: Traditional stack layout

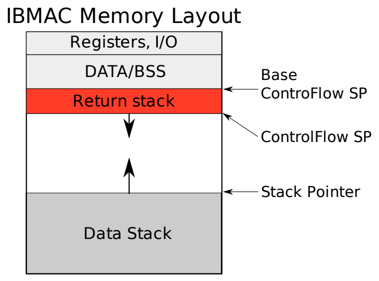


Figure 3: IBMAC stack layout. The Base control flow stack pointer is the only register that needs to be initialized in order to support IBMAC.

and modifications by other instructions should be prevented. Hence, restricting access to return addresses to `call` and `ret` instructions in hardware seems only logical. However in a normal stack layout, return addresses are interleaved to other types of data, making access controls difficult. In fact, such a fine grained access control would be slow and would lead to a considerable memory overhead, since all the words in memory that have to be protected would need to have an additional flag bit.

This is the main reason why we decided to modify the stack layout adding an additional *Return Stack*, specifically designed to store only return addresses. However, changing the memory layout could have lead to major compatibility issues. The principal design goal was to have a very simple hardware implementation, without extra memory requirement and focused on compatibility. The result is that IBMAC does not require modifications to the tool-chain and most binary libraries could be used without being rebuilt. IBMAC also improves software reliability as stack memory over consumption [16] can be detected at run-time so that a reboot or other actions can be performed (e.g. dedicated interrupt).

Finally we implemented IBMAC as an optional feature that can be activated for example with a write-once configuration register at boot[1]. With those constraints fulfilled and a proven implementation, we believe that this is a very realistic scheme with limited production costs and significantly increased security.

3.2 A separate return stack

In Figure 2 an architecture with a single stack is shown. While it is convenient to have a single stack, it makes it very difficult to protect the stored return addresses. We therefore implemented a modification to the instruction set architecture in order to support the use of two separate stacks: a *Return Stack* and a *Data Stack*. The return stack is used to store control flow information (return addresses) and the data stack is used to store regular data.

There are several different possible layouts in which those two stacks could be arranged in memory. The arrangement chosen in our implementation is depicted in figure 3. The first thing to note comparing figure 2 and 3 is that the data stack lies where the original single stack was. This is the best solution to maximize backward compatibility, as with this layout the data allocation on stack works in exactly the same way as before and no modifications to the compiler are necessary (e.g. to access local variables).

[1]This could be a *fuse* register on the AVR for example, as fuses cannot be modified without physical tampering.

The second thing to note is that the return stack and the data stack grow in opposite directions. This was done in order to optimize memory consumption, as with this layout no memory is wasted in comparison with the original stack layout. The fact that the return stack grows in the opposite direction does not hinder backward compatibility, as this stack is exclusively accessed in hardware by the modified `call` and `ret` instructions.

The third thing to note is that the return stack does not have any static limitation, but instead is only limited by the data stack. However this can also be a drawback as it those not leave room for an unbounded *heap*. In section 3.4 we discuss this problem in more detail.

3.3 Instruction Based Memory Access Control

The separate return stack layout presented in the previous section provides an easy way to separate control flow information from regular data allocated on the stack. However, it does not prevent modification and corruption of control flow information, but only makes it a bit more difficult as control flow data is not close to stack allocated buffers. Complex attacks could still be able to maliciously modify the return stack if an attacker is able to write data to an arbitrary memory location. This is possible for example with a double memory corruption (e.g.corrupt the pointer to an array and to further write data to this array), exploiting some format string vulnerabilities or is able to manipulate the stack pointer [10] to point to those memory regions.

This is the reason why an extra protection layer for the return stack is required. On a general purpose operating system this could be provided by a MMU. However, those are not available on such low end MCU. The reasons for that are multiple: first, those MCU are designed to be at a very low price range, each additional feature come at an increase of the silicon size and consequently increase the final manufacture price. Second, they are usually designed to execute monolithic application (often refereed to as firmware), therefore they do not require memory protection between different applications or the application and a kernel. The challenge is therefore to protect only the return stack at a very small cost, which is not the case with a complete Memory Management Unit.

Our hardware modification has been designed around the following considerations:

- only control flow related instructions will modify the control flow stack,
- the data manipulation instructions do not need to ac-

cess control flow information.

Given this observations it is possible to control memory accesses and decide whether to grant or refuse access to the return stack based on which instruction is performing the memory access. On the AVR we used, we identified only two instructions that needed to be able to access the return stack, namely the `call` and the `ret` instructions and their derivatives. The hardware implementation of these two instructions has been modified in such a way to set an internal flag to 1 whenever they are executed. When this signal is high memory access is granted to the control flow stack. If not, the system is rebooted (or could alternatively trow a dedicated interrupt).

3.4 Other design considerations

Dynamic memory allocation is one of the basic building blocks of modern operating systems and programing languages. However, it is often avoided on low cost embedded systems for the following reasons: first it is usually difficult to predict the worst case memory usage, which can quickly lead to memory exhaustion on these systems; second, memory fragmentation is a serious problem for architectures without Memory Management Unit. In fact, on architectures with a Memory Management Unit even if memory fragmentation happens in the virtual address space, it is always possible to defragment the physical memory, freeing large blocks of contiguous memory, in a transparent way for the application. This is not possible in the case of processors lacking a MMU because it would be necessary to keep track of all pointers and update them when the defragmentation process moves a contiguous memory block [2].

Usually on the AVR family of processors memory allocation is either performed statically i.e. global variables or when with dynamic allocation on the stack [3].

Nevertheless, if a heap is needed it is usually allocated within a fixed range of memory addresses for allocation. In such a case, the return stack can be made to start after the end of the heap, with risking overflows or memory waste.

4. IMPLEMENTATION AND DISCUSSION

4.1 Implementation

In order to validate our approach we implemented the changes to both a simulator and a soft core in a FPGA.

Implementation on simulator.

We modified the AVRORA [22] simulator in order to simulate the modified core, this made possible to run, by simulation, a complete platform with an Atmega128 [5] and a IEEE 802.15.4 [18] radio. We have been able to run unmodified TinyOS applications, for wireless sensor networks. The changes to AVRORA required modifications to only $0,4\%$ of the code (only 200 lines of code were changed while AVRORA simulator contains about 50,000 lines of code).

[2]It is possible to use double pointers, as done in the Contiki operating system. However, all access must be preformed with double de-reference, if an intermediate pointer is kept by the application and defragmentation occurs the memory might be corrupted by accessing an invalid address

[3]Variable memory allocation on the stack is possible using as GNU gcc's non standard *alloca* function

Implementation on a FPGA.

We implemented the modifications in a VHDL implementation of the Atmega103 core available at *opencores.org.* Although this micro-controller (MCU) version is discontinued, it is very similar to the Atmega128 and the modifications implemented are probably very similar to those required for an Atmega128. The modifications were made with changes of 8% of the VHDL source code (500 lines out of 6000). The resulting core was implemented on an *Altera* Cyclone II FPGA. The overhead in number of logical elements used (LUT) is of 9% (2323 LUT for the original MCU and 2538 LUT for the modified MCU). Although, this overhead might appear significant it is a non optimized implementation and as there is no extra memory requirements for its implementation, the overhead when implemented in an ASIC would probably be much lower.

4.1.1 Control flow modification operations

In the Atmel AVR core the program counter (PC) is not accessible as a general purpose register, instructions such as load and store cannot modify it. Therefore, there are only few instructions that can change the control flow, i.e. modifying the program counter or its saved value [4]. On the AVR the following instructions can modify the control flow:

- *Branch* and *Jump* (JMP) instructions update the control flow. However, as the destination address is provided as an immediate constant value, they are not vulnerable to manipulation and no return address is stored on the stack.
- *Call* and *Return* instructions use the control flow stack pointer to access the control flow stack. Those instructions will store or fetch the control flow instructions on the control flow stack.
- *Load* and *Store* instructions are prevented to alter the return stack, only access to data stack or other regions is allowed. The control flow stack and the data stack are checked to be non overlapping when a store is performed.
- *Calli* instruction takes a function pointer as parameter (from a register). This instruction is used for example in schedulers or object oriented code, in such a case an indirect call instruction is performed. If the attacker is able to modify the pointer (or register) before it is used by an indirect call instruction, he would be able to control one control flow change but not the following ones. However, solving this problem is out of the scope of this paper as it relates to protection of function pointers which can't be performed with this approach.
- Interrupts transfer the control flow to a fixed interrupt handler and the address of the instruction that was executed while the interruption occurred is saved on the control flow stack, in our modified architecture the return address is therefore protected as well.

One difficulty with the implementation of IBMAC is that the stack pointer as well as the control flow stack pointer

[4]This is not the case in all embedded cores, for example ARM cores have the PC as a regular register, therefore many instructions are able to alter the control flow.

Register name	Description	Atmega103 Address	Atmega128 Address
SP_CF_L	Control Flow Stack Pointer Low	\$00 (\$20)	\$46 (\$66)
SP_CF_H	Control Flow Stack Pointer High	\$01 (\$21)	\$47 (\$67)
SSCR	Split Stack Control Register (sec 4.1.2)	\$10 (\$30)	\$49 (\$69)
CF_SS_L	Control Flow Stack Start Low	\$02 (\$22)	\$55 (\$75)
CF_SS_H	Control Flow Stack Start High	\$03 (\$23)	\$56 (\$76)

(a) New register allocation for the additional registers.

Register name	Needs locking	Locking condition	Unlocking condition	Authorized modifications
SP	No	N/A	N/A	Any
CF_SP	Partial	After First Write	Reboot	Internal to CF instructions
CF_SP_Start	Yes	After First Write	Reboot	None
SSCR	Yes	After First Write	Reboot	None

(b) New registers locking logic

Figure 4: Stack configurations and control flow stack pointer description and additional locking logic

are 16 bit values and are modified with two instructions. Therefore, the update of the stack pointer is non atomic and its value can be temporally invalid. As a consequence it is not possible to enforce the constraints on stack pointers constantly. The solution we used is to enforce this constraint only when memory writes or reads are performed, with this approach the stack pointer can have a temporary invalid value when it is updated, without triggering an error.

4.1.2 Control flow stack configuration

The control flow stack needs to be configured before any control flow operation is used. It is activated from the “Split Stack Configuration Register” (*SSCR*). In order to prevent the attacker from maliciously change this register configuration, it is made “writable once per boot”: this configuration register is locked in hardware after the first write. The software (e.g. libc) is therefore responsible for setting this register during boot process. We use for this purpose the *init* sections provided in default linker scripts, so that the configuration is made as early as possible.

Memory layout stack memory areas configuration.

Compared to a traditional memory layout some configuration must be performed in order to enable the control flow stack and the memory access enforcement. For this purpose we implemented new configuration registers:

- *SSTACKEN* (Split STACK ENable) is a configuration bit which, when set, enables the split stack feature. It is part of the *SSCR* register.

- *CF_START* (Control Flow stack Start) is a configuration register used to fix the start of the control flow stack. It is automatically initialized from the libc to the end of the statically allocated memory (data/bss) therefore requires no user configuration.

- *CF_SP* (Control Flow Stack Pointer) is the control flow stack pointer. It is initialized with the same value than *CF_START* at boot and cannot be directly modified after initialization.

- *CF_STACK_configured* is an internal signal in our modified core. It is automatically set after control flow registers have been set up. It cannot be modified by

```
volatile uint16_t abssvar;
volatile uint32_t adatavar=10;

uint16_t factorial(uint16_t val){
    volatile local[10];
    if (val==1) return 1;
    else return val*myfact(val-1);
}

void factorial_with_smallalloc(){
    volatile uint8_t large[20];
    factorial(8);
}

void factorial_with_bigalloc(){
    volatile uint8_t large[200];
    factorial(8);
}

int main(){
    abssvar=10;
    factorial_with_smallalloc();
    factorial_with_bigalloc();
    return 0;
}
```

Figure 5: Example of a program that cause the stack to overflow

software and is reset when a reboot occurs. When this value is set any direct update of the *CF_START* and *CF_SP* registers are detected as possibly malicious modifications and therefore triggers a reboot. Without this an attacker could craft a fake stack and if he is able to modify the stack pointer (e.g. with an arbitrary memory write of two bytes) he could make it point to this fake stack. This fake stack would then be used as the legitimate stack.

These additional registers are described in Figure 4. In order to avoid conflict with existing peripherals devices or internal logic of the AVR cores the addresses of those configuration registers where chosen in the unused I/O registers addresses. The locking mechanisms that we implemented to prevent malicious manipulation of those registers are presented in Figure 4(b).

4.2 Evaluation

We evaluated the approach with different programs. Figure 5 shows an example program that has large stack memory usage. Two functions are present and are computing the factorial of a number with recursive calls. When the function with a larger array allocated on stack (*factorial_with_bigalloc*) is called a stack overflow occurs. Figure 6(a) shows the memory usage on an unmodified core, when the stack memory usage is too high the memory is corrupted and eventually unexpected behavior occurs. In this example program the stack pointer points to data and bss sections and later to IO Registers space, this results in erratic behaviours. On the other hand Figure 6(b) shows the resulting memory usage on an AVR core with split stacks and IBMAC. When the memory usage becomes too high the two stacks collide and the processor is rebooted by IBMAC. Similar results would be achieved if a malicious attempt to modify the control flow stack occurred.

4.3 Discussion

In addition to prevent control flow manipulation by abusing stack based buffer overflows and stack overflows, IBMAC also prevents malicious software present in the MCU to use return-oriented programming. In a MCU without IBMAC an attacker can use *return-oriented programming* for malicious purposes, such as maliciously hiding code memory [15, 23]. In order to use return-oriented programming a malicious program needs to write a *stack* containing both data and return addresses. While an attacker can craft such a stack on normal MCU, IBMAC prevents this as the malicious code isn't able to freely modify the return stack. Therefore, it is not possible to maliciously manipulate the control flow with return-oriented programming, even tough arbitrary code can be run on the device. In order to prevent this behavior, on a MCU where the attacker has full control, IBMAC needs to be permanently enabled. This can be performed using an irreversible configuration fuse. Without this the attacker would be able to restart the MCU on a modified program and deactivate the SSTACKEN configuration register.

Although our stack protection technique prevents control flow attacks as we described, it does not prevent all kind of software or logical attacks. Mainly, non control attacks [8] are not addressed because they do not rely on a change of the control flow but on overwriting adjacent variables. For example, a buffer overflow could be used to change the value of a variable used as a flag in an *if* statement. This in turn could be used for example to bypass specific controls in the program code.

Regarding the backward compatibility, while most software can run without modifications, the split stack scheme can make the implementation of features such as tasks with context switching and longjump / setjump difficult. Those features requires the software to be able to modify the stack and its control flow. If a kernel execution mode (or execution rings) were available, those features could be implemented safely. However, they cannot be implemented without major changes to the AVR core without the presence of such a privileged mode.

5. RELATED WORK

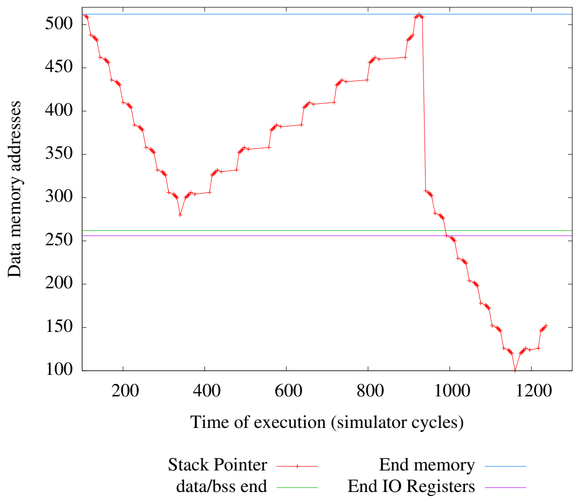
There is a wealth of different proposals on how to solve control flow vulnerabilities. In Control Flow Integrity Abadi et al. [1] propose to embed additional code and labels in the code, such that at each function call or return additional instructions a program is able to check whether it is following a *legitimate* path in a precomputed control flow graph. If the corruption of a return address occurs, that would make the program follow a non-legitimate path, then the execution is aborted as malicious action or malfunction is probably ongoing. The main drawback of the approach is the need for instrumentation of the code, although this could be automated by the compiler tool-chain, it has both a memory and computational overhead and thus might be infeasible on resource constrained devices.

Another possible solution was proposed in [6]. The authors propose to place a *canary* value between the return pointer and local function variables. The value of the canary value is set in the prologue of each function and is checked for validity in the epilogue. Canaries have been shown to have a number of vulnerabilities [2] and also require additional instructions to be executed at each function calls, thus introducing overheads.

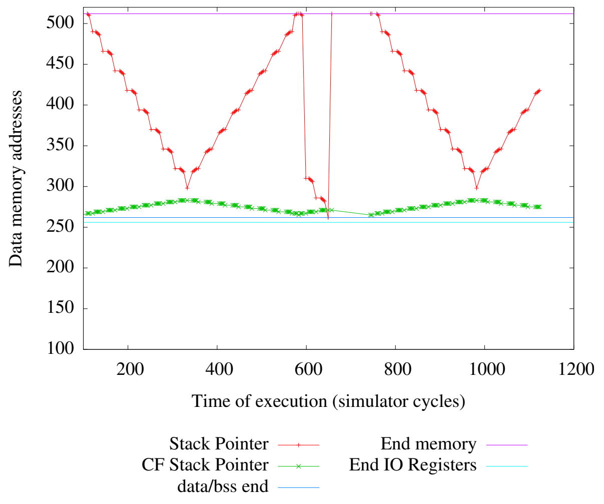
In [26] Yang et al. introduce a source to source transformation that translates traditional functions calls into a flat program. The transformation is similar to functions inlining without the usual code size overhead. The main limitation of this technique is that the transformation needs to be performed at source level and therefore requires a complete recompilation of the program. Therefore flattening cannot be applied to binary libraries or existing programs. Moreover, Interrupt handlers cannot easily be flattened as their call site and return address cannot be known in advance.

Address space layout randomization [20] can hinder control flow attacks. It is a technique where the base addresses of various sections (`.text`,`.data`,`.bss`, etc.) of a program memory are randomized before each program execution. Although, in [17] show that the effectiveness of address-space randomization is limited on 32-bit architectures by the number of bits available for address randomization. This problem would be even more severe on embedded systems that typically have a 8-bit or 16-bit address space.

In [13] the authors present StackShield that uses a compiler supported return stack. Where the compiler inserts a header and a trailer to each function in order to copy to/from a separate stack the return address from/to the normal stack. As this is implemented at the compiler level there is no backward compatibility, the programs need to be



(a) Execution *without* IBMAC. At point 1000 the stack is overflowing in the data/BSS section and later on the I/O register memory area.



(b) Execution *with* IBMAC enabled. When the return stack and the data stack collide (right after cycle 600), the execution of the program is aborted and restarted. This avoids memory corruption.

Figure 6: Comparison of the data memory layout during the execution of the program of Figure 5. In order to keep the example simple we ran the simulation with only 512 bytes of data memory address space.

re-compiled with this modified compiler. Moreover, as additional instructions are introduced there is non negligible a computation and memory overhead.

In [27] Younan et al. propose enables programs to use up to 5 different stacks and which separates them a compiler modification which that. While the approach appears similar to the one we present in this paper, the techniques used are different and they are adapted to different kind of systems. The multiple stacks technique introduced there is relying on guard pages to separate the stacks. This is possible only on hardware that has a MMU, without this it's impossible to make those guard pages and therefore provide some isolation between the stacks. Second, this approach to separate the stack in up to 5 different stacks would waste a lot of memory. On an AVR the stacks would need to be statically allocated and would therefore lead to an inefficient memory usage.

Similarly to our proposal in [25] the authors propose a return stack mechanism where dedicated `call` and `ret` instructions store and read control flow information from a dedicated stack. However the only guarantee for this return stack integrity is that is located far away the normal stack, which does not prevent modification of the return stack, it just makes it more difficult. Double corruption attacks [2] would allow an attacker to corrupt a data pointer first and then modify an arbitrary memory location on the return stack.

A number of systems already use a separate control flow stack like the PIC micro-controller (for example the pic16 [7]) or some AVR chips (AVR AT90S1200 [4]). However those solutions are not designed to improve security. They either allow direct modification of the hardware stack (vulnerable to double corruption) or have a limited stack stored inside the MCU (very limited call depth). For example the AVR AT90S1200 has a return stack supporting only 3 re-entrant routines, if more than 3 re-entrant interrupts or functions calls are performed the hardware return stack is corrupted.

The secure AVR [24] architecture is an evolution of the classical AVR code specifically enhanced for security. It is mainly used in smart cards and "smart" RFID chips. Unfortunately, only very few information are publicly available on those chips, as the manufacturer only provides short summary data sheets for the SecureAVR chips. We therefore cannot say whether their technique resembles the one described in this paper.

6. CONCLUSION

In this paper we introduced a split stack technique and an instruction based memory control that, when combined together, prevent malicious modifications of the control flow. This modified architecture was demonstrated as a modification of an AVR core. The solution presented is well suited for simple embedded systems that do not have a Memory Management Unit while introducing a very lightweight overhead in terms of a hardware implementation and, more importantly, has no extra memory usage. Therefore the presented technique could be implemented with a low extra cost. This technique completely prevents the modification of return addresses and prevent the attacker to craft a stack to in order to use techniques such as return-oriented programming. The technique was successfully implemented as a modification of an existing simulator as well as a soft core on a FPGA.

7. REFERENCES

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: 12th ACM conference on Computer and communications security*. ACM, 2005.

[2] S. Alexander. Defeating compiler-level buffer overflow protection. *Login;*, 30(3), June 2005.

[3] AMD. *AMD 64 and Enhanced Virus Protection*.

[4] Atmel, 2325 Orchard Parkway, San Jose, CA 95131. *8-bit Microcontroller with 1K Byte of In-System Programmable Flash , AT90S1200*, 2002.

[5] Atmel Corporation. Atmega128 datasheet. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.

[6] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *In Proceedings of the USENIX Annual Technical Conference*, pages 251–262, 2000.

[7] P. F. based bit and C. Microcontrollers. Pic16f688 data sheet.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium*, pages 177–192, 2005.

[9] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[10] G. Delalleau. Large memory management vulnerabilities; system, compiler, and application issues. CanSecWest 2005, May 2005. Presentation at CanSecWest, french article also published in the proceedings of SSTIC 2005 ”Vulnérabilités applicatives liées à la gestion des limites de mémoire”.

[11] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, October 2008. ACM.

[12] T. Goodspeed. Exploiting wireless sensor networks over 802.15.4. In *ToorCon 9, San Diego*, 2007.

[13] *StackShield*.

[14] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[15] F. F. Ralf Hund, Thorsten Holz. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Usenix security*, 2009.

[16] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *Trans. on Embedded Computing Sys.*, 4(4), 2005.

[17] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In B. Pfitzmann and P. Liu, editors, *Proceedings of CCS 2004*, pages 298–307. ACM Press, Oct. 2004.

[18] I. C. Society. Wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (wpans). `http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf`.

[19] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[20] The PaX Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[21] The PaX Team. Pax, 2003. http://pax.grsecurity.net.

[22] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN*, 2005.

[23] Undisclosed Authors. Revisiting code attestation in embedded systems. In *under submission*, 2009.

[24] S. D. VITO. *White Paper: Secure Microcontrollers for Secure Systems*. ATMEL, 11 2008.

[25] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks, 2002.

[26] X. Yang, N. Cooprider, and J. Regehr. Eliminating the call stack to save ram. In *To appear in LCTES 2009*, 2009.

[27] Y. Younan, D. Pozza, F. Piessens, and W. Joosen. Extended protection against stack smashing attacks without performance loss. In *Twenty-Second Annual Computer Security Applications Conference*, pages 429–438, 2006.