embSFI: An Approach for Software Fault Isolation in Embedded Systems*

Andreas Ruhland
fortiss GmbH
Munich, Germany
andreas.ruhland@tum.de

Christian Prehofer
fortiss GmbH
Munich, Germany
prehofer@fortiss.com

Oliver Horst
fortiss GmbH
Munich, Germany
horst@fortiss.com

***Abstract*—Software Fault Isolation (SFI) is a technique to sandbox software components based on transformation and checks on the assembly code level. In this way, software components can only access memory within specific fault domains. This paper presents embSFI, which applies selected SFI techniques to embedded systems in order to increase dependability and security, complementing or replacing a memory management unit. Our approach is designed to use SFI techniques which can be validated efficiently, even on embedded devices. Furthermore, we show that the overhead in performance is low, however depending on the scenario.**

I. Introduction

Software Fault Isolation (SFI) modifies code at the assembler level with instructions to guarantee that the code does not violate the restrictions required by the system. This can, for example, be instructions that ensure the target address of a memory instruction is valid. This concept was introduced more than 20 years ago by Wahbe et al. [1]. This approach does not target embedded ARM systems but some general concepts were used within embSFI. Another recent approach, NaCl [2], targets ARM hardware but requires virtual memory addresses and is not built for small embedded systems without full posix support. Concepts from both approaches are used to build embSFI which provides a fast online validation on embedded systems and a LLVM [3] compiler extension to generate suitable code. The concrete SFI patterns that make fast validation and execution on embedded systems possible were newly designed for embSFI. A more detailed description of this work can be found in [4]. In this paper software component and app are used interchangeably.

For secure and dependable critical systems it is important to keep the fault domains as small as possible. This means that should an error occur in one software component, other software components are unaffected. Small fault domains are one of the main motivations for microkernels, which are still extensively researched for critical systems [5]. There are various methods of creating fault domains, such as utilizing a memory management unit (MMU), virtualization or the aforementioned SFI.

* This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645119

EmbSFI uses SFI to enforce fault domains, and so we highlight the general advantages of using SFI for dependable critical embedded real-time systems in the following sections.

If a MMU or a memory protection unit (MPU) is not present within the embedded system, SFI can substitute their functionality. Should a MMU be available, SFI can still provide finer granularity on memory accesses. For example, with SFI it is possible to allow a software component to directly access only one byte in a memory area in a hardware memory mapped area. This can be necessary if the access to different hardware should be controlled to only grant access to specific software components. This could also be implemented by using a MMU and different execution levels and issuing a system call for access, but this approach would take control from the software developer. In addition, system calls can make the time analysis of software more difficult and make the system as a whole more complex. Because of that, SFI was used to decrease the context switch time for hardware drivers in microkernels [1].

When enforcing SFI, it is important to validate that every instruction in the binary is checked, and that the system only accepts binaries that fulfill the SFI invariants. This means that it is not possible to hide instructions, as can be done with self-modifying code. This is an important security feature as hardware cannot always be trusted and be susceptible to faults as shown with the rowhammering attack [6]. Within this attack, an instruction normally available in the user level of execution was used to change the expected behavior of the hardware. Through the use of SFI, such instructions can be disallowed. This is one reason why SFI is used inside the Chrome Browser to execute untrusted code [7].

The rest of the paper is organized as follows: Section II will describe the assembler properties of the embSFI system. The implementation of embSFI, including the compiler extension that generates code according to these properties, and the validator, which validates that the assembler code fulfills these properties, are discussed in Section III. An evaluation of the implementation with regards to its general limits and benchmark performance of the modified compiler are presented in Section IV.

II. Assembler Properties of embSFI

In this section, we describe the properties of the validatable assembler. embSFI combines concepts from [2] and [1] and adopts them to the ARMv7 architecture. These concepts are additionally made compatible for embedded systems without utilizing a MMU. A software component referred to as app, a consists of a code area and a data area. The code area of the app must be of a fixed size, C_a, that must be a power of two. The data area includes a variety of data such as heap and stack, and similarly must have a fixed predefined size D_a which is also a power of two. The execution environment for the app, similar to an operating system, can provide special functions that are callable from within the app. It is possible to allow the app to directly access specific memory areas outside of the private memory of the app. The concrete set of assembler properties were selected to be quickly validatable and allow for code that adheres to these properties to be easily generated. The following example code shows valid assembler code compared to code generated by an unmodified compiler:

Listing 1: without SFI

```
0
1  push{r4, sl, fp, lr}
2  add fp, sp, #8
3  movw r0, #0
4  movt r0, #0
5
6
7  bl 0 <print>
8  ...
```

Listing 2: with SFI

```
0  bfi sp, r9, #24, #8
1  push{r4, sl, fp, lr}
2  add fp, sp, #8
3  movw r0, #0
4  movt r0, #0
5  nop {0}
6  nop {0}
7  bl 0 <print>
8  ...
```

The different additional instructions in Listing 2 enforce the properties, no matter what the machine state is when entering this code.

We first describe the general properties of the modified code, and follow these with concrete patterns that are accepted by the embSFI validator.

A. embSFI approach

The main goal is to prevent the code of an app to jump somewhere out of its own code or to access memory outside of its own private memory area. However, there exist properties not directly connected to these goals, which are still necessary to be validatable.

1) Bundling: Bundling is a concept utilized by [2], that ensures every target of an indirect or direct instruction jump must be the start of a bundle. A bundle consists of multiple consecutive instructions with every instruction belonging to exactly one bundle. The binary size of the bundle B must be a power of two $B = 2^n$ for $n \in \mathbb{N}$. The bundle size is given by the environment and is the same for all apps. It can be assumed that the first instruction is located at the beginning of a bundle and all other instructions are consecutive and kept in the same order. Possible jump targets are normally function entries or branches of the app or selected function entries of the operation system. Assuming that hexadecimal addresses are used that address bytes, that a single instruction is 4 byte long as in the A32 Armv7 instruction set and that 4 consecutive instructions should always form a bundle, the resulting bundle size would be 16 bytes. This was the bundle size of Googles NaCl described in [2]. This means the start address of each code bundle should end with `0x0`. For example, `0xaaa0` and `0xfff0` are possible jump targets but `0xbbb1` is not.

Therefore, in Listing 2 it is not possible to jump directly to the push in line 1 and skip the sandboxing instruction in line 0.

2) Restricted Instruction Set: For embSFI only ARMv7 A32 instructions were considered which means no thumb instructions are currently possible like in [2]. Multiple instructions are forbidden completely, for example, store memory operations, which use a register as base address and as an index. All instructions that lead to undefined behavior or a fault are also not allowed.

3) Position independent code: The code of the app cannot assume that it will be located at a specific address during execution. This is also assumed by other binary loading concepts on embedded systems [8]. Therefore, the code must be position independent, which means that the code must either use program counter dependent addresses or use linker symbols instead of hard coded addresses.

4) Memory read and write: As realized in [1] the memory address where any instruction reads from or writes to must be inside the fixed size private data area of the app or an explicitly allowed memory address. The allowed memory access addresses do not change during the execution. The code segment of the app is not readable or writeable. This is a difference to [7] where read access to the code segment is allowed. In the SFI method proposed by embSFI the stack and the predefined app data are both in the same consecutive fixed size private data area. The size of the private data segment must be a power of two which means $D_a = 2^n$ for $n \in \mathbb{N}$. Also the start address of the data memory S_a area must be a multiple of D_a which means $S_a = D_a * n$ for $n \in \mathbb{N}$

5) Guard zones: As proposed in [1] guard zones are used to make certain accesses possible. For example in Listing 2 the push instruction manipulates 16 bytes and the base address is only guaranteed to be anywhere inside the private memory area of the app. This means that a small amount of memory around the data memory area must be reserved for the app. For example if the data memory areas goes form `0xaaaaaa00` to `0xaaaaaaff` and the guard zone G is 40 bytes big the address ranges `0xaaaaa9d8` to `0xaaaaaa00` and from `0xaaaaaaff` to `0xaaaaab27` have to be reserved exclusively. In difference to [1] no error is generated if the guard zones are accessed. They can contain for example parts of the stack.

6) Dedicated registers: Like proposed in [1] for MIPS dedicated registers can also be used in the ARM Architecture. In the register r9 the start address of the data segment is stored. It is shifted $log_2(D_a)$ bits to the right. This is only for an efficient implementation and is not used in [7] because the NaCl implementation relies on the memory management unit for separating code from data and it is not used in [1] because it is an optimization specific for certain ARM instructions as shown in the following sections. For example if the data segment starts at `0xdeadbe00` and is 256 bytes big the content of r9 is `0x00deadbe`. The code can always assume that r9 which is set by the environment contains the shifted address of the current data segment. Also the code is not allowed to use any instruction that could possibly change r9. As code and data memory areas are separated, the dedicated register r8 is used for the code and must always contain a valid jump address for the app. This means bundling must be respected and the size of the code segment. The app can assume that r8 always contains a valid jump address because it is set by the environment and enforced with the validation. The app can change r8 only if it is guaranteed that after the modification r8 again contains a valid jump address. This can be tricky and one way to do it is shown in Section II-B2.

B. Valid Patterns

To keep the validator very simple and small so it can be executed on an embedded system the patterns that are accepted are very simple. One simplification is to always look only at a bundle at a time. The validator will not accept code that applies to the principles but uses different patterns.

1) Memory access instructions: In this section possible valid A32 instruction sequences for memory accesses that can easily be validated are discussed. All instructions sequences are always in the same bundle. The most straightforward approach would be the direct access of data at a fixed address in the data memory area of the app. The address could be set by the linker.

```
mov r1,#dataaddress
ldr r2, [r1]
str r3, [r1]
```

If the linker replaces `dataaddress` with a valid constant address in the data segment of the app the validation of this code would be successful. The validator accepts several of move, load and store combinations.

If the memory access is the result of a complex calculation, the validation cannot track the value of the variable for example because the calculation is spread in multiple bundles. In this case, the register used for the access must be sandboxed as done by multiple other approaches [2] [1]. However, embSFI does a special optimized sandboxing for ARM using dedicated registers. We assume that the $D_a = 2^4 = 16$ and look at the following instruction sequence:

```
bfi r1,r9,#4,#28
ldr r3,[r1]
```

Because of the properties we can assume that r9 contains the address of the start of the data memory area S_a shifted $log_2(D_a) = 4$ bits to the right. Now the `bfi` instruction copies the bits 1 to 28 of r9 to the bits 4 to 32 of the register r1. In the instruction 4 is the start target bit and 28 is the number of bits copied [9, p. 2616]. Now we can assume r1 to be in the **data range**. As `ldr` loads 4 bytes the access is valid if the guard zone $G \geq 4$.

As ARM allows also constant offset the maximal constant depends on the guard zone size G. For example instead of `[r1]` using `[r1,#10]` requires $G \geq 14$.

The sandbox approach as described above has the disadvantage that a faulty app instead of immediately shutting down keeps on running using wrong values. An alternative approach which is also described in [1] is based on comparing the variable and jumping to an error handling routing if the address is out of range.

2) Branch instructions: In this section possible instruction sequences for branches that can easily be validated are discussed. It can again be assumed that all instructions are in the same bundle. During the validation, the absolute address of every single instruction is already known. So the A32 branch instruction `b` which uses a constant offset to the current program counter can easily be validated by calculating the constant target of the branch and checking if it is inside the apps code memory area and if it targets the beginning of a bundle.

For indirect jumps where the target address is normally stored in a register also valid instrumentations exist. We assume $B = 2^4 = 16$ and $C_a = 2^8 = 256$

```
bfc r1, #0, #4
bfi r8,r1,#0,#8
bx r8
```

The first instruction is a bit field clear which sets the bits 0 to 3 to zero [9, p. 2614]. Now the register r1 is **bundle aligned**. After that, the last 8 bits of the bundle aligned r1 are copied to the last 8 bits of the dedicated r8 register. This is valid because we can assume that r8 already contains a valid jump target and as the code segments starts 256 byte aligned and is 256 bytes big for all possible values of r1 r8 is still a valid jump target after the `bfi` instruction. This makes the `bfi` instruction valid. The instruction `bx r8`, which jumps to the address in r8, is always valid because r8 always contains a valid jump target as required by the dedicated register invariant.

For function calls normally the A32 instruction `bl` is used which pushes the address of the next instruction onto the stack and then jumps into the function [9, p. 2627]. This makes it easy to return to the callee. But also the return must follow the invariants which means that the target address to return to must be the start of an aligned bundle. This can be realized by filling up the bundle with operations that do nothing like `nop`. Assuming one bundle contains 4 instructions this would lead to:

```
nop
nop
nop
bl #function
```

This moving of the link instructions down at the bottom of a bundle was also used by Google and is already implemented in LLVM [2]. With these instrumentations a validator can easily check if the invariants are fulfilled by the code by only looking at one bundle at a time. Also by using always the same instrumentation, the validator gets simpler.

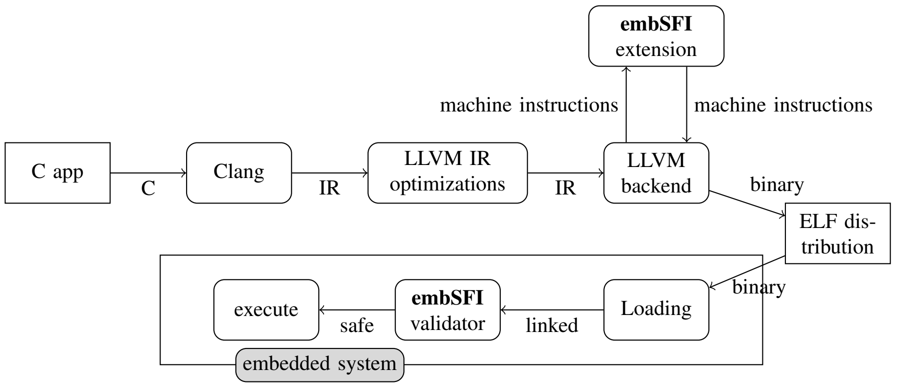

Fig. 1: Overview over the stages of instrumentation and validation

III. Implementation

The implementation of embSFI consists of different components like seen in Figure 1. In this section, the implementation of the different components will be described.

A. ELF loading

As binary format the ELF format was chosen because it is already used by other embedded systems [8], it is well documented for ARM [10] and supported by common compilers like `llvm` [3] and `gcc`. As embedded systems often have only a small amount of memory available other approaches exist which use smaller file formats that reduce the overhead of the ELF format [11]. Because of simplicity reasons embSFI uses the plain ELF format. The app must be a relocatable and not an executable. This is required so that the app can be linked easily to a specific hardware address during loading and does not depend on virtual memory addressing. Shareable ELF files would be harder to validate than already statically linked relocatables. To keep the code small and simple only the relocations `R_ARM_ABS32`, `R_ARM_CALL`, `R_ARM_MOVW_ABS_NC` and `R_ARM_MOVT_ABS` were implemented. There are over 100 different relocation types defined for the ELF format [10]. Some of them are not relevant for the SFI proposed here because they are used only in T32 ISA, which is not allowed so for the code generation these relocations were sufficient. The loader and linker included in embSFI requires the possibility to allocate aligned memory otherwise the allocation is very inefficient and takes up to double of the required memory like described in Section IV-A2.

B. Validation

The validator is implemented in C and so can be integrated into different execution environments or operating systems, which are often also written in C in the embedded environment. The validator expects the already linked code section and the data sections. The validator works directly on the binary machine code to optimize its memory footprint. The running time is $\mathcal{O}(number\,of\,instructions)$. In the evaluated examples described in Section IV the validation time was always below one second.

C. Compiler

As in [2] Clang was used for embSFI which already supports bundling. Other benefits of using clang are the decoupling of frontend language and output machine code. So the embSIF extension for LLVM can be also used with other languages that are not C. The embSFI extension for LLVM requires a parameter to specify the size of the required data area, which is fixed as explained in Section II

IV. Evaluation

The advantages of preferring SFI over MMU were already discussed in the Section I. These general advantages also apply to embSFI. In this section the general limits and drawbacks of embSFI will be discussed, followed by an evaluation of the performance of embSFI.

A. Limits and Drawbacks

1) No dynamic memory allocation: The embSFI principles require a fixed data size which does not change during the execution which means that the used memory cannot grow. Also all the required memory is reserved during the execution of the app. This is not a problem in for example high confidential real-time systems because there it should be guaranteed that the app is able to be executed and does not stop working in the middle of the execution because of a lack of available memory.

2) Allocated size: The size of the used memory can be the double of the original app because the code and the data memory areas should both have a size which is a power of two. So for example if an app uses 257 bytes of data the OS has to allocate $D_a = 512$ bytes. Therefore, the double size is the worst case scenario. Additionally the guard zones must also be allocated around the data area.

3) Aligned memory: The memory for the app must be allocated aligned. If no aligned memory allocation is available on the OS it could be required to allocate double of the needed D_a bytes of memory and then set the latest $log_2(D_a)$ bits of the aligned block to zero to get a valid S_a. This holds also for the memory used for the code.

4) Bigger code size: To ensure the invariants sometimes new instructions have to be added compared to the execution without SFI. The worst case size of the code can be multiplied with the instructions per bundle. So if every bundle consists of 4 instruction the code can be 4 times larger for SFI than without SFI.

5) No self-modifying code: As the code and the data segment are clearly separated it is not possible to use self-modifying code. This should not be needed in the most systems with high confidentiality because it makes code often more vulnerable to exploits. This restriction can also found in other SFI approaches like [2] and [1].

6) No function pointers to the OS: This is theoretically possible but not practical because it would make instrumentation much harder. The r8 register as described above can easily be used in the range of the own code of the app but not outside. One possibility of solving this is the use of trampolines like done in [2].

7) No usage of compiled code: It would be possible to transform already compiled code that does not fulfil all principles into instrumented SFI code. This can be done for example with binary rewriting tools like [12]. However, as the embSFI principles make assumptions about the dedicated registers r8 and r9 such a rewriting would be hard to implement or the resulting code would be very slow. So in general no compiled code without the source can be used.

8) No return exit: This means that when the app is finished it cannot do a normal return because this return would jump eventually into the OS where it is not allowed to jump. Either an OS function which is allowed for the app and terminates the app must be called or an endless loop must be inserted at the end of the app as last instruction.

9) Error detection: The current patterns guard memory access, which means that a memory access for an invalid address is changed to a valid memory access but an error is not detected. This was done because of performance reasons. It would also be possible to detect the error and handle an error routine provided by the OS. This would increase the instructions to add and so decrease the performance in regards to memory size and run-time.

B. Performance

As in ARMor [13] to evaluate the performance MiBench [14] was selected which is well suited for embedded system because it does not rely on OS functions like file handling and is implemented in C. The benchmark was executed on an ARM Cortex A-53 processor [15]. As compiler Clang with the modified LLVM [16] in the version 3.7 was used including the embSFI extension. The results were compared to binaries emitted by an unmodified Clang/LLVM 3.7. The source code to reproduce the benchmark results is available at `http://embsfi.de`.

1) qsort: The qsort small example from MiBench [14] was chosen to be evaluated because it heavily accesses the main memory. The test case uses 5000 strings and the qsort from the standard c library. The tested app consists of all parts necessary for the benchmark, which means that the qsort and all other used library functions are part of the app. The implementation from newlib [17] was used for that. The binary size overhead for all different optimization levels was around 90%. The overhead in the execution time is very different for the different optimization levels. Without optimization *O0* the overhead of the SFI variant is 143% for the time optimized variant *O3* the overhead is 82% and for space optimized *Os* variant the overhead is 39%.

2) bitcount: The bitcount large example from MiBench [14] was selected because it is not as memory bound as qsort.

The bitcount test uses different algorithms for calculating the number of bits in an integer. As bit counting is very fast every algorithm which is represented in an own function is called in a loop multiple times. This means some SFI overhead for calling a function and for returning from a function. Also some of the bit counting algorithms rely on lookup tables in the main memory which means memory access which has an overhead in SFI. The highest percentage of binary size overhead has the not optimized variant with 81%. Nevertheless, the differences between the variants are not very high as the speed optimized variant *O3* has 76% and the size optimized variant *Os* has 77%. One reason for the high overhead are the high number of small functions, which all need a safe return. Still the number of instrumented data accesses is in every optimization level more than twice as often used as the instrumented jump to a register.

The time is not as probably expected because in the case of the optimized binary size *Os* the SFI variant is nearly 9% faster than the non SFI variant. This could be because of a better cache behavior resulting from inserted `nops`. Still the both fastest variants are the *O3* variants without SFI and with SFI which has a run time overhead of 35%.

V. Related Work

Small fault domains in general can be achieved by sandboxing [18]. Sandboxing means that the untrusted application can only access restricted parts of the whole system to increase the safety and security of the system. It can be implemented in different ways for example with virtual machines (VM) or Software Fault Isolation (SFI). Process VMs can be implemented very efficiently also for small embedded systems which was demonstrated with Maté [19]. Another possibility which would not introduce any run-time overhead is proof-carrying code (PCC) [20]. This means that beside the code an encoded proof is delivered to the run time environment, which can check if that proof holds and if it fulfils specific requirements. Compared to PCC and VMs the approach of SFI which was introduced 1993 by Wahbe et al. [1] is simpler. The specific solution targets for example MIPS hardware but not ARM. Still some general concepts were used from that approach.

One more modern approach used by Google for their Native Client [7] to execute non trusted binaries is also based on SFI [2]. The overhead is as low as 5% for the ARM architecture. However, the system requires virtual memory and so the usage of a MMU. In addition, it does not target small embedded systems. There are also small SFI systems like ARMor [13] which are fully verified and optimized for the use in an embedded system. ARMor guarantees for example that all memory accesses are in a certain predefined area and that the program counter only goes to instructions in the predetermined control flow graph (CFG). ARMor uses the link-time rewriting framework DIABLO [12] for changing the binary and afterwards the higher order logic framework HOL [21] for verifying this. The verifying process is more complex than a simple online validation like in embSFI. Therefore, the verifying is done on separate machine and can take several hours.

Control Flow Integrity (CFG) can be enforced using methods similar to SFI and can be extended to full SFI solutions like shown by Abad et al. [22]. The system XFI [23] shows an approach for x86 systems. BGI [24] shows that such isolation can also happen at a byte granularity for x86.

VI. Conclusion

As demonstrated by embSFI, reducing the fault domains with SFI is achievable in embedded systems, including fast online validation. This also increases the security of systems because different separated software components cannot read private memory of other components. Even when a MMU is available, there are multiple reasons to prefer embSFI if a small run-time and memory size overhead is acceptable. In addition, embSFI is easy extensible. For example, if a software component should only be allowed access one bit directly in a hardware mapped memory this is achievable with SFI and not with an average MMU. The performance analysis of such an adoption for embedded ARM is left to future research. To create a secure system, the embedded validator must be implemented correctly. To guarantee that it would be necessary to proof the validator for example using HOL [21]. If this is achievable with acceptable effort needs still to be researched. The adoption and performance of embSFI to ARMv7 thumb and ARMv8 A64 instructions needs to be evaluated in the future. However, without these extensions embSFI can increase the dependability and security in embedded systems with a small effort and is easy to integrate in already existing C solutions, which consist of separable software components.

References

[1] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/173668.168635

[2] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *19th USENIX Security Symposium*, 2010, pp. 1–11. [Online]. Available: http://code.google.com/p/nativeclient/

[3] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, March 2004, pp. 75–86.

[4] Andreas Ruhland, "Secure run-time binary loading in a trusted real-time environment," Master's thesis, Technische Universität München, Germany, January 2016.

[5] A. Tanenbaum, J. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, vol. 39, no. 5, pp. 44–51, May 2006.

[6] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 361–372, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665726

[7] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*, May 2009, pp. 79–93.

[8] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182810

[9] *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile Beta*, Arm ddi 0487a.g (id070815) ed., ARM Limited, 7 2015.

[10] *ELF for the ARM ® Architecture*, Arm ihi 0044e, current through abi release 2.09 ed., ARM Limited, Nov. 2012.

[11] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, "Dynamic linking and loading in networked embedded systems," in *Mobile Adhoc and Sensor Systems, 2009. MASS '09. IEEE 6th International Conference on*, Oct 2009, pp. 554–562.

[12] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *ISSPIT*, Dec 2005, pp. 7–12.

[13] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: Fully verified software fault isolation," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: ACM, 2011, pp. 289–298. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038687

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[15] ARM Ltd, "Cortex-a53 processor - specification," January 2016, http://www.arm.com/products/processors/cortex-a/cortex-a53-processor.php.

[16] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[17] J. Johnston and T. Fitzsimmons, "The newlib homepage," *URL http://sourceware. org/newlib*, 2016.

[18] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *USENIX Security Symposium Volume 6*, ser. SSYM'96. Berkeley, CA, USA: USENIX Association, 1996, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267569.1267570

[19] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 85–95, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/635506.605407

[20] G. Necula, "Proof-carrying code. design and implementation," in *Proof and System-Reliability*, ser. NATO Science Series, H. Schwichtenberg and R. Steinbrüggen, Eds. Springer Netherlands, 2002, vol. 62, pp. 261–288. [Online]. Available: http://dx.doi.org/10.1007/978-94-010-0413-8_8

[21] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press, 1993.

[22] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity - principles, implementations, and applications." Association for Computing Machinery, Inc., November 2005. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=69217

[23] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, November 2006. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=64368

[24] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, Inc., October 2009.