μShield

Configurable Code-Reuse Attacks Mitigation For Embedded Systems

Ali Abbasi[1(✉)], Jos Wetzels[1], Wouter Bokslag[2], Emmanuele Zambon[3], and Sandro Etalle[1,2]

[1] Services, Cyber Security and Safety Group, University of Twente, Enschede, The Netherlands
{a.abbasi,sandro.etalle}@utwente.nl, a.l.g.m.wetzels@student.utwente.nl
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
w.bokslag@student.tue.nl
[3] SecurityMatters BV, Eindhoven, The Netherlands
emmanuele.zambon@secmatters.com

Abstract. Embedded devices are playing a major role in our way of life. Similar to other computer systems embedded devices are vulnerable to code-reuse attacks. Compromising these devices in a critical environment constitute a significant security and safety risk. In this paper, we present μShield, a memory corruption exploitation mitigation system for embedded COTS binaries with configurable protection policies that do not rely on any hardware-specific feature. Our evaluation shows that μShield provides its protection with a limited performance overhead.

Keywords: Embedded · Code reuse · Heuristics · ARM

1 Introduction

From critical infrastructure to consumer electronics, embedded systems are all around us and underpin the technological fabric of everyday life.

The rise of the Internet-of-Things has seen a widespread proliferation of so-called 'smart devices' with everything from fridges to smoke detectors and door locks being fitted with a small computer communicating with its environment.

Just like any computer, these devices have vulnerabilities that can be exploited by attackers. The sheer number of embedded devices and their pervasiveness in our lives makes them an attractive target. Attackers are now starting to focus on embedded devices is demonstrated by the recent attacks on DYN network, and the release of the MIRAI botnet [14]. The result is that manufacturers that historically did not have to worry about the security of their embedded products now face a challenging situation: The security measures developed in the last 20 years for general-purpose computers are hard to apply to embedded systems. This is so for a number of reasons which include the fact embedded systems are very diverse from each other in terms of computational resources.

Z. Yan et al. (Eds.): NSS 2017, LNCS 10394, pp. 694–709, 2017.
DOI: 10.1007/978-3-319-64701-2_55

In this paper, we focus on protecting embedded devices from memory corruption and code-reuse attacks such as buffer overflow, heap exploitation, use-after-free and Return Oriented Programming (ROP). These attacks exploit an important class of vulnerabilities because of the fact that embedded software development is dominated by the C language (around 66%) [8,18]. Indeed, in the recent security literature, we find a number of approaches addressing memory corruption attacks in embedded systems [11,19,27].

Here, we depart from the previous approaches because we want to take into consideration from the start the following three constraints that we argue being of crucial importance in the ecosystem of embedded systems:

Firstly, embedded systems are extremely heterogeneous in terms of processing power (which can be very low), the available resource and responsiveness requirements. This implies that a memory protection solution must address this issue, by providing flexible protection based on the performance specification of the embedded system.

Secondly, vendors of various embedded equipment tend to procure third party software only available in Commercial Off-The-Shelf (COTS) binary form, without access to source code of this software.

Thirdly, the hardware landscape of embedded systems is much more diverse than that of general-purpose computers, and we cannot expect that embedded systems already in production be retrofitted with upgraded hardware. In general, one cannot rely on the presence of any particular hardware, hardware-facilitated functionality or hardware-specific features for a memory corruption mitigation approach in embedded systems.

In this paper, we introduce μShield an open source [2] code-reuse mitigation system for embedded binaries. μShield addresses the performance and requirements diversity (the first constraint) by providing configurable protection policies, that can be tailored to the specific system. Also, μShield protects COTS binaries without relying on the control-flow graph and can work with a hardware-agnostic cryptographically secure shadow stack.

To the best of our knowledge, no memory corruption mitigation approach for embedded systems addresses all limitations mentioned above. Finally, for evaluation of μShield we choose ARM architecture due to its wide application in the embedded world.

1.1 Our Contributions

The main contributions of our work are the followings:

- **Configurable Policies:** μShield provides configurable protection policies. The user can specify different levels of protection (depending on an overhead-security trade-off made by them).
- **Hardware agnostic:** μShield is not relying on the presence of any special hardware, hardware-facilitated functionality or hardware-specific features.

- **Cryptographically secure parallel shadow stack:** with our shadow stack implementation we present the, to the best of our knowledge, first parallel shadow stack for ARM and the first hardware-agnostic adoption of cryptographically-enforced protection for shadow-stacks in general.
- **Stack frame integrity walker:** we propose a new lightweight, coarse-grained backward-edge CFI heuristic which works by walking stack frame chains and checking all saved return addresses for control-flow integrity. It imposes minimal overhead while being hardened against known attacks against coarse-grained CFI techniques.
- **Performance evaluation based on the worst-case scenario:** we evaluate μShields performance and memory overhead in several worst-case scenarios instead of the average case scenario. We show that our basic level of protection consistently manages to stay below the 1% overhead while our advanced level of protection manages to stay below 15% overhead in most scenarios.

2 Background

In the last decade the research community suggested two different approaches to address memory corruption vulnerabilities both in general-purpose computers and embedded systems. The approaches are the followings:

- Behavior-based Heuristics: there have been various proposals to detect control-flow hijacking, by leveraging execution behavior of exploit characteristics, such as heap-spraying detection [16], detection of specific payload behaviors such as external library loading or stack pivots.
- Control-flow Integrity: CFI is a technique placing restrictions on control-flow transfers to make runtime control-flow conform (with various degrees of accuracy) to intended program control-flow. The seminal work by Abadi et al. [1] proposed a fine-grained analysis and enforcement scheme which unfortunately incurred high overhead. This has lead to the proliferation of CFI proposals seeking to address performance overhead through various trade-offs. However CFI systems also have their own limitations as we describe it in Sect. 2.1.

2.1 Applicability of Related Works for Our Constraints

Since one of the constraints we described for embedded systems protection was COTS support, in Table 1 we list the initial selection of COTS binary supporting solutions against several of the other criteria imposed by our environment as outlined in Sect. 1. In the table, the term *hardware-agnostic* indicates whether a solution relies on features specific to particular hardware (e.g., Intel's LBR) or not. The term *CFG reliance* indicates whether a solution requires CFG extraction from the protected binary in question in order to function. The term *bypassed* shows that bypasses for the work have been constructed under attacker models equal to or weaker than the one presented in Sect. 2.2 in either academic work or practical exploitation. With regards to performance overhead, we rely

Table 1. Platform applicability

Solution	Hardware-agnostic	CFG Reliance	Bypassed	Worst. overhead
kBouncer [24]	No (LBR)	No	Yes	6%
ROPdefender [13]	Yes	No	No	200%
DROP [9]	Yes	No	Yes	530%
ROPstop [20]	Yes	Yes	No	19.1%
PathArmor [33]	No (LBR)	Yes	No	27.3%
BinCFI [35]	Yes	No	Yes	42%
CFCI [36]	Yes	No	No	83%
MoCFI [11]	Yes	Yes	No	1106.8%
O-CFI [22]	No (Intel MPX)	Yes	No	11%
Lockdown [26]	Yes	No	No	273%
CET [25]	No (Intel CET)	No	No	Unreported

on the worst overhead reported by the authors of each work, or measured by Burow et al. [6].

As can be seen in the Table 1, none of the surveyed solutions meets all the criteria. One interesting conclusion that can be drawn is that there seems to be a triangular trade-off between hardware-agnosticism, security and performance overhead, with the best-performing solutions either being hardware-facilitated or lacking in offered security.

2.2 Threat Model

Our attack scenario consists of control-flow hijacking memory corruption attacks under a *minimum system security baseline*. Minimum system security baseline consists of deploying existing readily available, exploit mitigation techniques named as NX, ASLR, Stack canaries and Full RELRO to our protected application.

We assume powerful attacker model in which the attacker has an arbitrary info-leak primitive (i.e., can read from arbitrary locations in memory) and a vulnerability allowing them to overwrite control-flow elements (e.g., return-addresses, function pointers). While most modern security mechanisms assume the attacker cannot (arbitrarily) read memory, under our attacker model the attacker can bypass them (e.g., by using the info-leak to bypass stack cookies and ASLR [3]) provided she can to construct a code reuse payload to bypass Non eXecutable (NX) memory protection.

***-Oriented Programming (XOP).** We use the term "*-Oriented Programming" (XOP) to refer to exploitation techniques making use of code reuse in general. Taxonomically, code reuse attacks can be divided into Return-Oriented

Programming (ROP), Jump-Oriented Programming (JOP) [4], Call-Oriented Programming (COP) [7]. Given our target architecture (ARM) (where there are no dedicated call, return or jump instructions but rather direct and indirect branching instructions) we can conflate ROP, JOP and COP into the single category of XOP with little problems.

3 μShield Design

3.1 Design Overview

μShield offers two levels of security, *basic* and *advanced*, on a per-application configurable basis. μShield consists of three core components:

1. **Setup Module**: The setup module checks whether the system meets minimum baseline requirements and harvests instrumentation points from a given application for the Runtime Protection Module (RPM) configuration file.
2. **Kernel Protection Module** (KPM): The KPM offers the *basic* protection level in the form of behavior-based heuristics and coarse-grained backward-edge CFI. It is implemented as a kernel module hooking a variety of security-sensitive system calls. Upon hooks invocation the KPM execute its heuristic.
3. **Runtime Protection Module** (RPM): The RPM offers the *advanced* protection level in the form of fully-precise backward-edge CFI and coarse-grained forward-edge CFI. It is implemented as an LD_PRELOAD library instrumenting function prologues and epilogues to implement a shadow stack (for backward-edge CFI) and code pointer calls to implement dynamic function call validation (for relaxed forward-edge CFI).

Figure 1 illustrates the deployment of μShield on a target system.

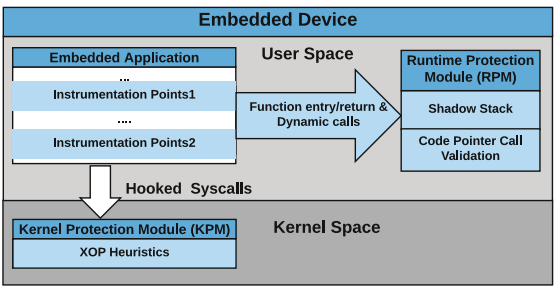

Fig. 1. High-level illustration of a μShield deployment.

3.2 Detection Mechanisms

The detection mechanisms of the KPM and RPM are based on behavior-based heuristics and CFI policies. Generally speaking, the former offer inferior security coverage but impose less overhead, while the latter offer better security coverage with more overhead. For this reason, we adopt behavior-based heuristics in our basic protection component (the KPM), to allow users to enable only the KPM for the most lightweight variant of μShield. Given the strict overhead constraints for embedded systems, the lack of access to target applications source-code (and hence our limited ability to extract accurate CFGs), we choose coarse-grained CFI in our advanced component (the RPM).

3.3 Kernel Protection Module

Minimum security baseline is essential for KPM. The restrictions imposed by the minimum security baseline force the adversary to exploit vulnerabilities using a limited set of patterns which is well known to the KPM. This allows us to employ a lightweight monitoring approach, which triggers inspection only at specific points during the execution of an application using a limited set of heuristics.

Stack Frame Integrity Walker. The coarse-grained backward-edge CFI in the KPM is implemented in the form of a stack frame integrity walker. As part of calling conventions, functions get allocated a local stack frame containing, among other things, a return address to the caller. In order to facilitate debugging and error reporting many applications require the ability to unwind stack frames, that is, to walk a chain of nested function calls backwards from the current frame all the way to the top of the stack. The most common and stable way to facilitate this is through the presence of frame pointers, which are present in every local stack frame and constitute a pointer to the previous stack frame, resulting in a linked list that can be walked upwards. It is thus possible to inspect a local stack frame, walk the chain upwards and inspect all return addresses along the way. Our walker does just this and, for every return address encountered, decides whether it is valid, meaning it is preceded by a Branch-with-Link (BL) instruction (the ARM equivalent of a call). As per calling convention, every function call has to return to an address preceded by such an instruction and violation of this indicates CFI has been subverted. In this case, we raise an alert. In addition, return-addresses are not allowed to point to stack or heap memory. This principle is known as branch-precedence [15] and has been part of other coarse-grained CFI solutions.

However our approach is completely different from the work in [15]. Firstly existing branch-precedence heuristics only check the current stack frame for a branch-preceded return address. μShield instead validates the return address of the entire stack frame chain. Secondly, heuristics based on [15] can be bypassed by means of so-called trampoline gadgets (also known as Call-Ret [12], Call-Site [17] and Invocation gadgets [30]). To address this issue, some CFI solutions use length-based heuristic gadget classifiers and raise an alert if more than N

sequences of less than M instructions were spotted (i.e. N gadgets). But such classifiers can be bypassed by using heuristic breakers such as long-NOP and termination gadgets [7,12]. Our work instead seeks to identify trampoline gadgets by checking if a return address has a call/indirect-branch-with-link and return-type instruction within N instructions from the gadget start. If so, we mark it as a trampoline gadget. Once more than M trampoline gadgets have been detected we raise an alert. We found $N = 5$ and $M = 2$ as ideal values that do not produce false positives. Evading this heuristic would require attackers to use less than M trampoline gadgets, exposing them to regular branch-precededness checks in the process. Note that this heuristic constitutes only our lightweight protection for devices which simply cannot afford fine-grained CFI.

3.4 Runtime Protection Module

Backward-Edge CFI Using Shadow Stack. μShield uses full parallel shadow stack [10] for the program stack meaning that all stack operations of the program is synchronized with our shadow stack. In μShield the function call handler pushes the return address on the shadow stack while the function return handler checks the top of the shadow stack against the return address.

Forward-Edge CFI. In addition to the backward-edge CFI offered by the shadow stack, we include in the RPM a (relaxed) forward-edge CFI mechanism for dynamic code pointer calls which uses a function prologue validation heuristic. The heuristic validates whether the call destination is a valid function prologue. The coarse-grained nature of this measure means that some degree of security is traded for performance and applicability. Since forward-edge CFI relies on precise approximation of the intended application CFG and our environmental constrains impose a binary-only solution without reliance on CFG extraction, we opt for a coarse-grained CFI that considers any function prologue (but only valid function prologues) a valid control-flow destination for dynamic code pointer calls.

3.5 Theoretical Analysis of CFI Mechanisms

Using the theoretical taxonomy provided by Burow et.al. [6] we obtain the following security qualifications for the CFI aspects of our solution:

- KPM: Backward-Edge (D), CF.1, SAP.F.0, SAP.B.1
- RPM: Backward and Forward-Edge (D), CF.1/CF.5, SAP.F.1a, SAP.B.2

The comparison provided by Burow et.al. [6] can give an overview of μShield compared to other CFI solutions. We can conclude our KPM provides minimalistic backward-edge CFI while our RPM provides minimalistic forward-edge CFI but highly precise backward-edge CFI (which, in addition, is not hampered by hardware limitations).

4 μShield Implementation Details

4.1 Setup Module

Setup module consists of the following sub-system:

- Harvesting function prologues and epilogues: We identify all functions in the target binary. We search prologues and epilogues for the register-saving and register-restoring instructions which respectively save and restore the return address from the stack. These addresses are added to a configuration file for use by the RPM.
- Harvesting code pointer call sites: We identify all functions in the target binary. Within the function body, we search for register-relative Branch-with-Link (i.e., BLX Rx) instructions, which is how ARM represents dynamic code pointer calls. These addresses are added to a configuration file for use by the RPM.

We implement the setup module in our prototype using two different underlying frameworks: IDAPython which is built on top of IDA Pro and the Angr [31] framework. In both cases, identification of functions is done heuristically without requiring reliable CFG extraction and both frameworks can work with COTS binaries without debugging information.

4.2 Kernel Protection Module

Syscall Hooking. The KPM time of check is set to (a subset of) syscall invocations. We do this by fetching the system call table address and replacing the entries to be hooked with the addresses of our hook functions, while storing the original addresses so they can be called by the hook functions.

Stack Frame Integrity Walker. The walker is illustrated in Fig. 2. It walks the chain of stack frames from the current stack frame up to the topmost frame and, along the way, checks whether the return addresses contained within them

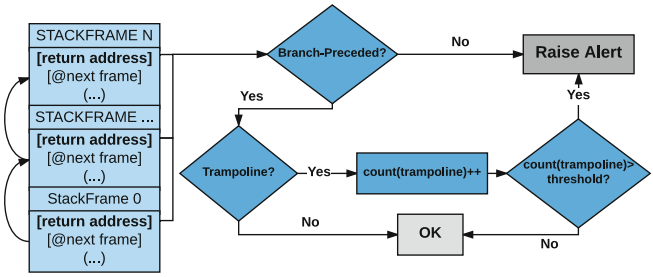

Fig. 2. Stack frame integrity walker

are valid. If it encounters an invalid return address, an alert is raised. Furthermore, it counts the number of return addresses which qualify as *trampoline gadgets*(comprising Call-Ret Pair [12] and Call-Site [17]) if the count exceeds a certain threshold, it raises an alert. This part of our heuristic was specifically designed to address the attacks where attacker crafts payloads consisting of gadgets which are branch-preceded, thus illegitimately qualifying as valid return-sites [12,17]

We consider a return-address *valid* if and only if it does not point to the stack or heap and is Branch-with-Link-preceded. Also, we consider an address a *trampoline gadget* (or dispatcher/call-site gadget [17]) if and only if it contains, within threshold of N instructions from its start, an indirect Branch-With-Link instruction (or semantic equivalent) followed within M instruction by any indirect branch.

Maximum walking depth. We walk the stack frame chain upward up until a threshold depth of N (where N is larger than the deepest function nesting we can reasonably expect) after which we terminate. We do this in order to prevent attackers from executing a denial of service attack through crafting self-referential stack frames which would cause an infinite loop in kernel space.

Chain-walking complications. Walking the chain of stack frames can be tricky depending on system circumstances. The ideal scenario is one where binaries are compiled with frame pointer support (what we name FP-compliance) in which case we can simply take the Frame Pointer register (FP), look up the stack frame, and walk a linked list backwards to the top.

Developers have argued against omission of frame pointers for decades [5] since they are required for efficient stack trace calculations. FP-compliance up to recently was also mandatory for compliance with the Embedded Application Binary Interface (EABI). Alternatively μShield can use binary debugging information or static backward data flow reconstruction using static analysis (although suboptimal) to complete the chain-walking.

4.3 Runtime Protection Module

Instrumentation. Our instrumentation approach is done completely native to the RPM and does not rely on any pre-existing frameworks, since these either do not offer ARM support (e.g., static instrumenting like DynInst, PEBIL) or they come with significant overhead (e.g., dynamic instrumentation like Valgrind, DynamoRio, PIN).

Our approach consists of identifying the function's main routine start point and instrumenting it to set up the shadow stack and subsequently taking all the instrumentation points identified by the setup module and instrumenting them (with detour hooks) redirecting control flow to our shadow stack handler and code pointer call handler routines. These routines are implemented in ARM assembly and are designed to be as lightweight as possible to limit overhead.

Shadow Stack. In order to provide fully-precise backward-edge CFI we implement a lightweight checking parallel shadow stack as illustrated in Fig. 3. We instrument the first instruction of the program's main routine to call a shadow stack setup handler, which allocates a memory area which will serve as a dedicated shadow stack, located at a fixed offset from the stack-pointer. As the program executes, the shadow stack synchronizes with the regular stack thus avoiding the need to walk the entire shadow stack to check for valid return addresses (as is the case with regular shadow stacks). This parallel approach allows us to avoid problems plaguing implementations of traditional shadow stacks [13,34]. While our current prototype does not support multi-threading, there is no reason μShield cannot be extended to support multi-threading by using multiple dedicated shadow stacks for each thread.

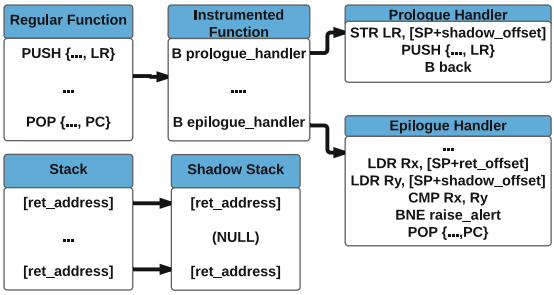

Fig. 3. Checking, parallel shadow stack

We instrument all function prologues and epilogues to detour to shadow stack prologue and epilogue handlers (fully implemented in assembly). The prologue handlers take the return address (stored in the Link Register (LR)) and write them to the shadow stack (i.e., the memory address at fixed offset from the current stack pointer).

Cryptographically-Enforced Shadow Stack Variant. Since most shadow stack proposals and implementations do not deal with the issue of securing the shadow stack itself, Mashtizadeh et.al. [21] have proposed Cryptographically-enforced CFI (CCFI) which stores the keyed Message Authentication Code (MAC) tag of a combination of the shadow stack address and original return address to the shadow stack rather than the original return address itself. The shadow stack prologue handler calculates the tag as `*tag = MAC(return_address | shadow_address | ..., secret_key)*` and stores it to the shadow stack and the epilogue handler simply takes the intended return address of a function, calculates the corresponding tag, compares it against the stored tag on the shadow stack and raises an alert upon mismatch. The shadow stack address is included with the return address in order to prevent attackers from swapping two tags on the shadow stack (similarly to replay protection).

With this approach, even an attacker capable of manipulating the shadow stack itself cannot hijack control-flow without being able to forge a MAC entry, a problem which is reducible to the cryptographic security of the MAC.

However, the CCFI approach in [21] does not meet our criteria since it is neither hardware agnostic (it relies on the x86 AES-NI extension) nor binary COTS compatible (it requires source-code access). We adapted CCFI to meet our criteria by implementing a binary-COTS compatible version in our shadow stack prologue and epilogue handlers using a customized software-only implementation of the lightweight Chaskey-8 [23] MAC algorithm for ARM consisting of only 166 instructions. In addition, we omit any information beyond return address and shadow stack address from the MAC to improve performance. Given that the security of the MAC rests on the secrecy of the key it is important it does not leak to the attacker. As such, the key is stored in a large register (eg. NEON or VFP on ARM or XMM on x86) unused by the program or otherwise determined to never leak to the program state.

Code Pointer Call Validation. To provide a form of forward-edge CFI we instrument all *code pointer calls* that is, all indirect register-relative branches, and check whether they point to a valid function prologue. This is a "relaxed" form of CFI, since it does not restrict an attack in redirecting hijacked code pointers but it does restrict them to target valid function prologues rather than any XOP gadget.

We leverage the fact that our minimum security baseline guarantees all functions are compiled with stack cookie support. This means that each function prologue will contain a sequence of instructions setting up the stack cookie and storing it on the stack which we will use as an instruction signature for validating function prologues. Thus, given a target code pointer call destination address, we can check whether we find the above instruction sequence within threshold instruction bound N and, if not, raise an alert. The number of false negatives here is minimal as the first instruction in the sequence consists of the loading of a rarely referenced static address located in the program images .bss segment (the stack cookie storage address) followed by a dereference and storing instruction.

In addition we impose that there are no branch instructions of any kind in between the above instructions, to prevent attackers from targeting potential gadgets that, for whatever reason, happen to conform to the above form or gadgets located less than N instructions before a valid function prologue.

The Code pointer call validation restricts function calls to legitimate function prologues rather than only intended ones. As such it rules out targeting of arbitrary gadgets but leaves room for an attacker to swap arbitrary function calls. An attacker wishing to extend this ability into crafting an actual gadget chain would need to use Entry-Point Gadgets (EP-Gadgets) [17] which consist of a sequence of instructions starting at a legitimate function entry point and end with an indirect branch. EP-Gadgets thus begin at allowable destinations for control transfers. While an attacker could use EP-gadgets to bypass our code pointer call validation heuristic they would need to craft an entire

chain consisting only of EP-gadgets since all code pointer calls are instrumented with our forward-branch validation code. In addition, our instrumentation of function prologues and epilogues means that within such an EP-gadget chain any executed prologue needs to be matched with a corresponding epilogue in order to prevent shadow-stack mismatches from raising an alert which implies EP-gadgets can only be executed as chains of fully-executed functions rather than the cobbled-together segments that usually constitute gadgets. As such we consider practical exploitation of this weakness to be highly complicated if not practically infeasible in most cases. Given the extremely low overhead impact of this measure we consider the above weakness to be acceptable, especially in the light of the security offered by our backward-edge CFI.

5 Evaluation and Discussions

5.1 Performance Evaluation

We performed the overhead evaluation on a Raspberry Pi 1 Model B+, which features a 800 MHz single-core ARM1176JZF-S CPU and 512 MB RAM, running the Raspbian Jessie Lite Linux distro with Linux kernel 4.1. In order to reduce system noise which could interfere with benchmarking we ensured tests were run on a "barebones" system with no services or applications apart from core system processes running alongside the tests.

Due to historical reasons, most authors working on CFI tend to use the SPEC [32] CPU benchmarking suite to measure performance overhead. However, we cannot adopt this approach for evaluating μShield due to several reasons.

First, existing work in the area of memory corruption mitigation tends to measure its overhead against applications representative of an average case usage scenario. This is unsuitable for our purposes, since an overhead indication in such average-case scenarios tends to wildly vary from the overhead experienced in worst-case outliers scenarios.

Secondly, our hardware (a Raspberry Pi) does not satisfy the minimum hardware requirement of SPEC. SPEC requires 1 GB of RAM in a 32bit CPU while our Raspberry Pi does provide only 512 MB or memory. As alternative to SPEC benchmarks, in addition to the applications we selected to represent worst-case scenarios, we choose the SciMark2 [29] scientific computing benchmarking suite, since its functionality was integrated as part of SPEC but did (unlike the full SPEC suite) meet the requirements of our platform.

Constrained Overhead Test. We refer to our overhead testing suite as the Constrained Overhead Test (COT). The COT consists of the following components.

- **SciMark2** [29]**.** A benchmark for scientific and numerical computing designed by NIST. SciMark has been integrated as part of SPEC.

- **lshw.** A linux tool that gathers information about the hardware present in the system. lshw executes a series of syscalls in order to obtain low-level information about hardware capabilities which makes it suitable as a test for the frequent invocation of our KPM heuristics.
- **primes.** A demonstration program shipped with GMP (GNU Multiple Precision arithmetic library), that computes a list of all primes between two given numbers using a prime number sieve, with a large number of function calls in every execution.
- **nqueens.** A simple program recursively solving the n-queens problem commonly used as part of CPU performance benchmarking suites such as the Phoronix Test Suite [28].

Overhead Figures. Each component of the COT was tested with four different configurations of μShield to get insights into the performance overhead imposed by individual and combined components.

Per configuration, each benchmark was run 50 times so as to minimize random bias. Memory overhead was measured for the RPM in all applicable configurations but not for the KPM (due to technical complications). However, since the KPM does not allocate any memory on the heap nor allocates any stack variables of significant size, we can consider its imposed memory overhead negligible. Table 2 reports an overview of CPU overhead measurements for the entire COT, while Table 3 reports memory overheads.

The results show that for the benchmarking suite `SciMark2` we remain below 4.7% performance overhead at all times. The results also show that for the selected worst-case scenarios the imposed CPU overhead tends to stay below 14.4%. When only basic level security is enabled (in the form of the KPM) CPU overhead always stays equal to or below 0.5% with the bulk of the overhead of full protection being due to the RPM backward-edge CFI (in the form of its

Table 2. Worst case CPU overhead overview for KPM, KPM plus RPM Forward-Edge (FE), KPM plus RPM Backward-Edge (BE) and KPM plus full RPM protections.

Benchmark	No protection	KPM	KPM + RPM FE	KPM + RPM BE	KPM + Full RPM
SciMark2 overhead	27.105	27.240	27.089	28.359	28.199
	-	0.50%	0.06%	4.63%	4.04%
lshw overhead	16.946	16.955	17.009	19.260	19.373
	-	0.06%	0.38%	13.65%	14.33%
primes overhead	10.126	10.139	10.169	10.698	10.835
	-	0.13%	0.42%	5.65%	7.00%
nqueens overhead	4.105	4.106	4.105	4.112	4.112
	-	0.02%	0.00%	0.16%	0.17%

Table 3. Memory overhead for KPM, KPM plus RPM Forward-Edge (FE), KPM plus RPM Backward-Edge (BE) and KPM plus full RPM protections.

Benchmark	No protection	KPM	KPM + RPM FE	KPM + RPM BE	KPM + Full RPM
SciMark2 overhead	10791	10799	10810	10859	10830
	-	0.07%	0.17%	0.63%	0.36%
lshw overhead	26547	26475	26729	27475	27382
	-	−0.27%	0.68%	3.49%	3.14%
primes overhead	10835	10860	10858	10872	10858
	-	0.23%	0.21%	0.35%	0.22%
nqueens overhead	10833	10826	10858	10806	10795
	-	−0.06%	0.24%	−0.24%	−0.35

shadow stack). This result does, however, strengthen our argument for a modular solution design, in which vendors using applications closer to such a scenario of extreme recursion could decide to opt for dropping the RPM backward-edge CFI which, as shown in Table 2, results in dropping virtually all of the imposed overhead. Measured memory overheads, as outlined in Table 3, is negligible in all cases with an observed maximum of 3.49%.

6 Conclusions and Future Work

In this paper, we presented a new code-reuse mitigation system for resource constrained embedded devices named as μShield. μShield considers the general constraints imposed by embedded systems such as performance limitations, lack of fully featured hardware or COTS binaries. Our evaluation shows that μShield can detect memory corruption attacks defined in our scope and have acceptable performance overhead under worth case scenarios. Based on our evaluation, we can argue that despite the limitations in embedded systems, it feasible to have a protection mechanism for devices with a different level of resources.

Finally, the configurable protection policies and non-intrusive detection approach of μShield paves the way for addressing stricter availability requirements such as hard real-time for the embedded systems in the future.

Acknowledgement. The work of the fifth author has been partially supported by the Netherlands Organization for Scientific Research (NWO), through SpySpot project (no. 628.001.004)

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: ACM Conference on Computer and Communications Security (CCS) (2005)
2. Abbasi, A., Wetzels, J., Zambon, E.: μShield: host-based detection for embedded devices used in ICS environments. https://github.com/preemptive-FP7/uShield
3. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D.: Hacking blind. In: IEEE Symposium on Security and Privacy (2014)
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2011)
5. Brucker, J.P.: ARM: deprecate old APCS frame format. http://lists.infradead.org/pipermail/linux-arm-kernel/2016-February/404969.html
6. Burow, N., Carr, S.A., Brunthaler, S., Payer, M., Nash, J., Larsen, P., Franz, M.: Control-Flow Integrity: Precision, Security, and Performance. arXiv (2016)
7. Carlini, N., Wagner, D.: Rop is still dangerous: Breaking modern defenses. In: USENIX Security Symposium (2014)
8. Cass, S.: The 2015 top ten programming languages. IEEE Spectrum **20** (2015)
9. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: Drop: Detecting return-oriented programming malicious code. In: International Conference on Information Systems Security (2009)
10. Dang, T.H., Maniatis, P., Wagner, D.: The performance cost of shadow stacks and stack canaries. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2015)
11. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.R.: Mocfi: A framework to mitigate control-flow attacks on smartphones. In: Symposium on Network and Distributed System Security (NDSS) (2012)
12. Davi, L., Lehmann, D., Sadeghi, A.R., Monrose, F.: Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: USENIX Security Symposium (2014)
13. Davi, L., Sadeghi, A.R., Winandy, M.: Ropdefender: a detection tool to defend against return-oriented programming attacks. In: ACM Conference on Computer and Communications Security (CCS) (2011)
14. Dobbins, R.: Mirai IOT botnet description and DDOS attack mitigation. Arbor Threat Intell. **28** (2016)
15. Fratrić, I.: Ropguard: runtime prevention of return-oriented programming attacks. (2012). https://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf
16. Gadaleta, F., Younan, Y., Joosen, W.: Bubble: a javascript engine level countermeasure against heap-spraying attacks. In: International Symposium on Engineering Secure Software and Systems (2010)
17. Goktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy (2014)
18. Group, U.E: Embedded markets study (2015). https://webpages.uncc.edu/jmconrad/ECGR4101-2015-08/Notes/UBM%20Tech%202015%20Presentation%20of%20Embedded%20Markets%20Study%20World%20Day1.pdf
19. Habibi, J., Panicker, A., Gupta, A., Bertino, E.: DisARM: mitigating buffer overflow attacks on embedded devices. In: Qiu, M., Xu, S., Yung, M., Zhang, H. (eds.) NSS 2015. LNCS, vol. 9408, pp. 112–129. Springer, Cham (2015). doi:10.1007/978-3-319-25645-0_8

20. Jacobson, E.R., Bernat, A.R., Williams, W.R., Miller, B.P.: Detecting code reuse attacks with a model of conformant program execution. In: International Symposium on Engineering Secure Software and Systems (2014)
21. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: cryptographically enforced control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (2015)
22. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque control-flow integrity. In: Symposium on Network and Distributed System Security (NDSS) (2015)
23. Mouha, N., Mennink, B., Herrewege, A., Watanabe, D., Preneel, B., Verbauwhede, I.: Chaskey: an efficient MAC algorithm for 32-bit microcontrollers. In: Joux, A., Youssef, A. (eds.) SAC 2014. LNCS, vol. 8781, pp. 306–323. Springer, Cham (2014). doi:10.1007/978-3-319-13051-4_19
24. Pappas, V.: kBouncer: efficient and transparent ROP mitigation (2012). http://www.cs.columbia.edu/vpappas/papers/kbouncer.pdf
25. Patel, B.: Intel release new technology specifications to protect against ROP attacks. https://blogs.intel.com/evangelists/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/
26. Payer, M., Barresi, A., Gross, T.R.: Lockdown: dynamic control-flow integrity. arXiv preprint arXiv:1407.0549 (2014)
27. Pewny, J., Holz, T.: Control-flow restrictor: compiler-based CFI for IOS. In: Annual Computer Security Applications Conference (ACSAC), pp. 309–318 (2013)
28. Media, P.: Open-Source, Automated Benchmarking (2016). http://www.phoronix-test-suite.com/
29. Pozo, R., Miller, B.: SciMark 2 (2016). http://math.nist.gov/scimark2/
30. Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., Holz, T.: Evaluating the effectiveness of current anti-ROP defenses. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 88–108. Springer, Cham (2014). doi:10.1007/978-3-319-11379-1_5
31. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
32. Standard Performance Evaluation Corporation: SPEC's Benchmarks (2005). https://www.spec.org/benchmarks.html
33. van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical context-sensitive CFI. In: ACM Conference on Computer and Communications Security (CCS) (2015)
34. Zhang, M., Qiao, R., Hasabnis, N., Sekar, R.: A platform for secure static binary instrumentation. ACM SIGPLAN Notic. **49**(7), 129–140 (2014)
35. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: USENIX Security Symposium (2013)
36. Zhang, M., Sekar, R.: Control flow and code integrity for cots binaries: an effective defense against real-world rop attacks. In: ACM Conference on Computer and Communications Security (CCS) (2015)