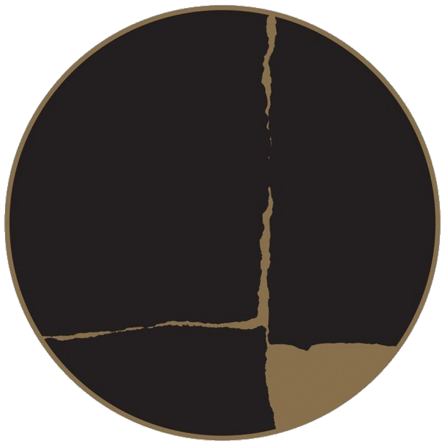
KINTSUGI

Identifying & addressing challenges in embedded binary security

JOS WETZELS



Supervisors:
Prof. dr. Sandro Etalle
Ali Abbasi, MSc.

Department of Mathematics and Computer Science
Eindhoven University of Technology (TU/e)

June 2017

Jos Wetzels: *Kintsugi,* Identifying & addressing challenges in embedded binary security, © June 2017

To my family

Kintsugi ("golden joinery"), is the Japanese art of repairing broken pottery with lacquer dusted or mixed with powdered gold, silver, or platinum. As a philosophy, it treats breakage and repair as part of the history of an object, rather than something to disguise.

— [254]

ABSTRACT

Embedded systems are found everywhere from consumer electronics to critical infrastructure. And with the growth of the Internet of Things (IoT), these systems are increasingly interconnected. As a result, embedded security is an area of growing concern. Yet a stream of offensive security research, as well as real-world incidents, continues to demonstrate how vulnerable embedded systems actually are.

This thesis focuses on binary security, the exploitation and mitigation of memory corruption vulnerabilities. We look at the state of embedded binary security by means of quantitative and qualitative analysis and identify several gap areas and show embedded binary security to lag behind the general purpose world significantly.

We then describe the challenges and limitations faced by embedded exploit mitigations and identify a clear open problem area that warrants attention: deeply embedded systems. Next, we outline the criteria for a deeply embedded exploit mitigation baseline. Finally, as a first step to addressing this problem area, we designed, implemented and evaluated `µArmor` : an exploit mitigation baseline for deeply embedded systems.

When you want to know how things really work,
study them when they're coming apart.

— William Gibson

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisors, Prof. dr. Sandro Etalle and Ali Abbasi, MSc., for their assistance and support throughout the process of researching and writing this thesis. Sandro's guidance helped me structure the rather voluminous body of research produced by this thesis while Ali's continuous feedback and brainstorming helped this work achieve its current depth.

Secondly, I am grateful to my family, without whom none of this would have been possible, for their unconditional love and support.

Finally, I am thankful to my friends for their moral support and to the many people from the information security field and the outer reaches of the internet who, over the years, have continued to inspire me.

Jos Wetzels
Eindhoven University of Technology (TU/e)
June 2017

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

GP	General Purpose
CPS	Cyber-Physical Systems
MCU	Microcontroller Unit
SoC	System-on-a-Chip
RTOS	Real-Time Operating System
MMU	Memory Management Unit
MPU	Memory Protection Unit
OEM	Original Equipment Manufacturer
ODM	Original Device Manufacturer
ICS	Industrial Control System
ESP	Executable Space Protection
ASLR	Address Space Layout Randomization
TRNG	True Random Number Generator
(CS)PRNG	(Cryptographically Secure) Pseudorandom Number Generator

Part I

EMBEDDED BINARY SECURITY

1 INTRODUCTION

1.1 PROBLEM STATEMENT

Embedded systems are everywhere. From consumer electronics, networking equipment and critical infrastructure to vehicles, airplanes, military hardware and medical equipment: they underpin the technological fabric of society [88, 237, 240–242]. Over the years embedded systems have become increasingly interconnected [13], a trend that is exacerbated by the rise of the so-called *'Internet of Things (IoT)'* which has been projected to grow from roughly 5 billion [102] deployed devices in 2015 to anywhere between 20 [102] and 50 billion [139] in 2020.

Contrary to its popular image, the IoT is not just limited to *smart home* and *wearable* consumer electronics. While most people are familiar with such gadgets, the real drive behind IoT growth comes from enterprise applications [121, 245]. Some examples are asset tracking, production line monitoring and automation in the *industrial IoT* [431], *smart grid* [335] and smart meter deployment in the energy sector and streamlining of lighting, parking, surveillance and public infrastructure optimization in *smart cities* [358].

The growing interconnectedness of embedded systems has made security an increasingly important concern [13, 101, 202, 203]. Not only because the systems themselves are interesting targets for attackers but also because many of these systems weren't designed to be connected to networks in the first place. And with the growth of the IoT these concerns are set to spread to many different sectors. Such concerns are not merely theoretical as real-world incidents involving embedded systems have shown. Nor are they limited to one particular group of systems or industry vertical: incidents occur everywhere from the infamous `Stuxnet` attack against nuclear facilities in Iran [413], recent attacks against Ukrainian power grid [178, 278, 323] and the hacking of smart TVs by intelligence agencies [70] to the formation of massive botnets consisting of compromised IoT devices [145]. And with new attacks on embedded systems in everything from cars [36, 308] to medical equipment [189, 451] being demonstrated, the importance of embedded security is virtually self-evident.

Yet embedded systems security is generally seen as lagging behind what we've come to expect of our general purpose (eg. desktop and server) systems [202, 203, 205, 421, 422]. Embedded *'binary security'* in particular is an area where exploitation of vulnerabilities is significantly easier than on general purpose systems. This is exemplified by a 2016 incident where a previously unknown group calling themselves the *"Shadow Brokers"* released a cache of exploits which they claimed belonged to the supposedly state-sponsored *"Equation Group"* [78] threat actor. Among this cache were a set of exploits for high-end firewall equipment of multiple manufacturers [5], none of which had to bypass any exploit mitigations.

However, despite the general perception of embedded systems binary security as lagging, there is a lack of quantitative and qualitative research providing insight into the current state-of-the-art and its gap areas.

1.2 RESEARCH GOAL & QUESTIONS

Research Goal: The goal of this work is to identify the state-of-the-art in embedded operating system binary security and contribute to improve the security of embedded systems, in particular against memory corruption attacks.

The following **research questions** have to be answered in order to achieve the stated research goal:

RQ-1: What would a minimum exploit mitigation baseline for embedded systems look like?

RQ-2: What is the current state-of-the-art in embedded operating system exploit mitigations in terms of adoption, dependency support and implementation quality?

RQ-3: What are the gap areas and open problems within the current state-of-the-art and what are the challenges underlying them?

RQ-4: Given the clearest gap area identified, what would an effective solution look like and what criteria should it meet?

1.3 CONTRIBUTIONS

This work makes the following, to the best of our knowledge, novel contributions:

1. We establish a minimum exploit mitigation baseline for embedded systems and outline its security criteria and hardware- and software dependencies.

2. We perform the first *quantitative analysis* of the state of embedded exploit mitigation adoption and support for software- and hardware dependencies. This analysis shows that embedded systems lag behind general purpose ones significantly in this regard.

3. We perform the first *qualitative analysis* of the exploit mitigations and secure random number generators of three embedded operating systems: `QNX`, `RedactedOS` and `Zephyr`. This analysis shows their implementations to be flawed in various ways. We responsibly disclosed the vulnerabilities we uncovered in the course of this analysis to the vendors in question and collaborated in drafting fixes.

4. In order to explain the results of our analyses, we provide the first systematic identification of the challenges embedded exploit mitigation adoption efforts face. We then identify two major open problems and outline the criteria for their solutions.

5. We propose, implement and evaluate µ`Armor` : the first exploit mitigation baseline design for constrained *deeply embedded systems*, thereby addressing the identified open problems.

1.4 OUTLINE

Part i provides a background on embedded systems and binary security in Chapter 2 and presents an embedded exploit mitigation baseline in Chapter 3. **Part ii** presents our analyses of the state of embedded exploit mitigations in the form of a quantitative analysis in Chapter 4, a qualitative analysis in Chapter 5 and a discussion of challenges, open problems and criteria in Chapter 6. **Part iii** presents our work on µ`Armor` and discusses its design in Chapter 7, its implementation in Chapter 8 and its evaluation in Chapter 9. **Part iv** contains an overview of related work in Chapter 10 and discussion, conclusions and future work in Chapter 11. Finally, **Appendix A** contains supplementary data such as tables, benchmarking information and code snippets.

2 BACKGROUND

2.1 BASIC EMBEDDED CONCEPTS

The term *'embedded systems'* covers a wide range of devices used for a myriad of different purposes. They can be as simple as a single pressure sensor node or a digital alarm clock or as complicated as flight management systems, medical imaging equipment or military weapons systems.

Embedded & General-Purpose Systems

An **embedded system** is *"a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car. Contrast with general-purpose computer"* [402].

This is contrasted with a **general purpose (GP) system** which is *"a combination of computer hardware and software that serves as a general-purpose computing platform. PCs, Macs, and Unix workstations are the most popular modern examples"* [402].

Due to the incredible diversity of embedded systems, it is impossible to say what a 'typical' embedded system looks like. They can range from high-end systems with a user interface to constrained *deeply embedded systems* engaging only in *machine-to-machine (M2M)* communications as part of a *Cyber-Physical System (CPS)*.

Deeply Embedded & Cyber-Physical Systems (CPS)

Deeply embedded systems [423] are those systems on the most highly integrated and constrained end of the embedded spectrum and serve dedicated, single purposes (eg. collecting sensor data, basic processing or automation, etc.). These systems tend to use resource-constrained *microcontrollers* which allow for low production cost. Deeply embedded system often lack user interfaces and tend to be either *bare metal* or run extremely minimal operating systems. They are generally not (dynamically) programmable once the program logic has been burned into ROM and require either device programmers or bootloader-assisted firmware reflashing.

Cyber-Physical Systems (CPS) are a *"class of engineered systems that offer close interaction between cyber and physical components"* [442] characterized by deep integration between physical and software components that are *"typically designed as a network of interacting elements with physical input and output instead of as standalone devices"* [131]. Examples of CPS include automotive, avionics and aerospace systems, industrial control systems and 'smart grids', robotics systems, medical monitoring and robotic surgery systems, military and defense systems, etc.

As a result, embedded systems hardware can range from simple 8-bit *microcontrollers* to a *System-on-a-Chip (SoC)* outfitted with a multicore 32- or 64-bit microprocessor.

Microprocessors, Microcontrollers (MCUs) and System-on-a-Chip (SoC)

Microprocessors incorporate CPU functionality on a single integrated circuit (IC) (or at most a small number of them), reducing the cost of processing power. **Microcontroller Units (MCUs)** essentially integrate a CPU along with memory and peripherals on a single IC. This high degree of integration allows for size and cost reductions that make MCUs ideally suited for embedded deployment.

The term **System-on-a-Chip (SoC)** refers to systems with a high degree of integration of all components of a computer or other electronic system. The difference with MCUs is often one of degree, with MCUs generally being actual single-chip systems while the term SoC is often used as hyperbole indicating a high degree of but not completely single-chip integration. Typical SoCs consist of an MCU, microprocessor or digital signal processor (DSP) core and memory, peripherals, external interfaces (eg. USB, Ethernet, etc.) or power management functionality.

Some embedded systems run an operating system while others are only outfitted with a piece of *bare metal* (ie. OS-less) application firmware. Embedded operating systems themselves come in many different flavors too, from variants of popular general-purpose OSes to tiny *real-time operating systems (RTOS)* and from *monolithic* kernel architectures to *microkernel* designs.

Real-Time Operating System (RTOS)

Real-Time Operating Systems (RTOS) are *"designed specifically for use in real-time systems"* [402], the latter being *"any computer system, embedded or otherwise, that has timeliness requirements"* [402]. That is to say: those systems which must be able to guarantee a response within specified time constraints with varying degrees of 'hardness'. The 'harder' these constraints, the more serious missing a deadline is considered. As such real-time systems are more about deterministic and predictable response times than fast ones [344]. Real-time requirements are particularly important in safety-critical applications such as flight control software, automotive brake controls or industrial manufacturing processes.

Monolithic & Microkernel Architectures

As opposed to **monolithic kernels** where *"all functionality provided by the OS is realized within the kernel itself"* [344], a **microkernel** *"reduces the services provided by the kernel dramatically by putting all services, which are not essentially necessary for the microkernel, into user space as isolated processes. (...) The service processes typically behave like servers of the client-server model. To use such a service an application needs to send a message with a service request to the service which receives the request, completes the request and sends back a response message to the client application. (...) The big advantage of microkernels against monolithic kernels is the clear separation of services from the kernel itself making the kernel a very small piece of software that provides a better fault isolation and can be maintained more easily than a monolithic kernel. The fault isolation prevents crashing the whole system. (...) The price we have to pay for the better structuring and fault isolation is that we get a high amount of interprocess communication through message passing and a high amount of context switching"* [344].

While the high-end embedded operating systems found in smartphones or certain networking equipment are often based on Linux-, BSD- or even Windows and as such are familiar to people used to the general purpose world, the *library-based operating systems* aimed at the low-end of the embedded spectrum are quite different as illustrated in Figure 1.

Library-Based Operating Systems

Library-based operating systems implement the OS as a collection of libraries which are linked together with the application code into *"one single executable which is executed in one single address space. Therefore no loader is required to dynamically load applications at run-time, by this minimizing the operating system code. Another advantage of a library-based RTOS and the execution in a single common address space is that system calls can be simply implemented as function calls. Thus no context-switches are required when calling an operating system function. This is often more efficient and less time consuming as a full context switch with address space changes when having an RTOS implemented as a kernel in a separated address space"* [344]. A disadvantage of library-based OSes is the lack of safety and security that comes with running all application code and kernel code in the same address space as a result of the lack of memory separation.

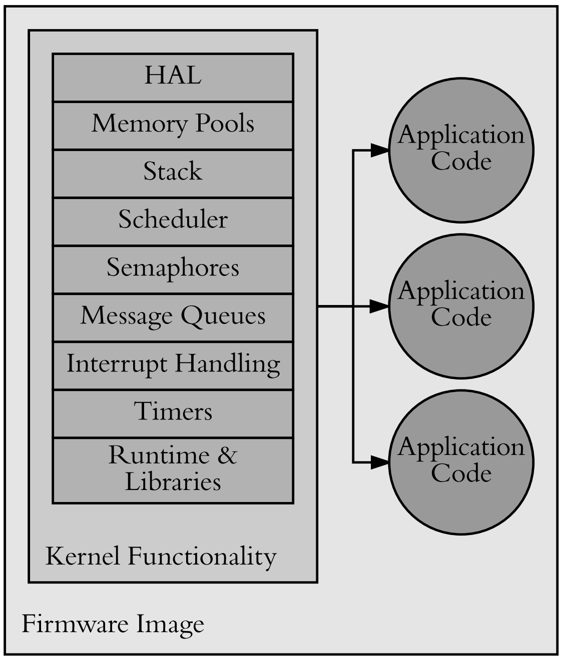


Figure 1: Library-Based Operating System Example

2.2 EMBEDDED SECURITY THREAT LANDSCAPE

The world of embedded systems is incredibly diverse and spans many industry verticals. As such, rather than establishing a single comprehensive overview of the threat landscape we will outline some of the key differences between the embedded and general purpose worlds:

1. **Security Attention**: Security has been a serious concern in a minority of industries dealing with embedded systems (eg. payment cards, video game consoles, PayTV solutions, etc.), but this has usually been the result of the fact that security issues in such devices directly threaten corporate revenue streams, intellectual property or the core business model. Since this is not the case for most other industries, attention to the security of embedded systems has continued to structurally lag behind what we've come to expect of general purpose systems. A recent joint report by the *European Union Agency for Network and Information Security (ENISA)* and semiconductor manufacturers *Infineon*, *NXP* and *STMicroelectronics* speaks of a *"market failure for cybersecurity and privacy"* [56], a sentiment which is echoed by Bruce Schneier [202].

   The *ENISA* report observes that there is *"no basic level, no level zero defined for the security and privacy of connected and smart devices"* [56], a troubling observation considering the increasing connectivity and corresponding exposure of embedded systems [13]. This is compounded by a common 'confusion' between *safety* and *security* requirements. While the prior is a core requirement of many embedded systems and enforced by regulations and certifications, it does not cover the latter which deals with *actively malicious* subversion of a system. This was illustrated in one recent study [88] where 22% of the surveyed designers of Internet-connected safety-critical systems indicated security was not a design requirement.

2. **Security Concerns**: The security concerns of embedded systems are often quite different from those in the general purpose world as well. A 2017 embedded safety & security study [88] found the three primary security concerns to be *product tampering*, *theft of data* and *theft of intellectual property*, all of which relate more to the company which designed the product than to the users. This is despite the fact that security threats to embedded systems are growing more serious with the involvement of government-sponsored hacking [17] rather than the traditionally more restricted attackers such as pirates or competitors.

When attacking desktop or server systems, attackers often attempt to obtain users' documents or sensitive financial information. In embedded systems, however, attacker goals tend to be more device-specific because of their single-purpose nature. An attacker might try to sabotage a *cyber-physical system* [281] (causing electrical blackouts or car crashes), tamper with measurements (smart meter fraud), bypass authentication procedures (building access control) or device restrictions (pirated entertainment media), conduct surveillance (network equipment or smart home hacking) or steal intellectual property (code extraction and reverse engineering). Sometimes the attacker goal is only indirectly related to the embedded system in question, such as when connected embedded systems are used to pivot into corporate networks [64, 125] or launch *Distributed Denial-of-Service* attacks [126].

Such attacks can involve anything from flipping a few bits somewhere in memory or extracting a single cryptographic key to obtaining full code execution; in some cases, attackers might have physical access to the device in question (especially when *users themselves* are potential adversaries). This opens up a wide range of attacks [8] ranging from abusing exposed debugging functionality [93] or side-channel and glitching attacks [80, 170] to more invasive attacks involving chip decapping and optical reverse-engineering [80, 236, 283]. A more exhaustive survey of threats and concerns can be found in the *ENISA "Hardware Threat Landscape and Good Practice Guide"* [55].

3. **Ecosystem Differences**: Embedded 'ecosystems' are characterized by a number of particular challenges, addressed in Chapter 6, around development practices, resource constraints, system requirements and limitations that are very different from the general purpose world. With respect to security these ecosystem differences also translate to difficulties in patching (addressed in Section 2.3), host- and network-based intrusion detection [193, 205, 333, 395, 397, 433] and forensics [12, 119, 120, 427]. Finally, as observed in a recent Barr Group study [88], the sheer *diversity* of embedded systems in terms of hardware, software and applications prohibits the emergence of a one-size-fits-all security solution.

The embedded devices used in certain industries pose a bigger risk than those in others. Networked, safety-critical devices in particular pose a major risk and within that group a handful of systems is associated with more than two-thirds of the risk [88]: *medical devices*, *industrial control systems* and *automotive systems* followed by *consumer electronics* and *defense & aerospace systems*.

Below we present a brief overview of some of the practical attacks, not all of which are relevant to *embedded binary security*, that have been demonstrated to affect these systems in order to illustrate how serious the situation is. For a more exhaustive overview of the threats we refer to the work in [129, 281, 330, 449].

Medical: Medical devices are often intimately coupled to lethal risk but despite this practical attacks have been demonstrated on infusion pumps [82, 115, 189, 314], pacemakers and ICDs [320, 328], ECGs [424], surgical robots [451] and bionic arms [6]. While real-world incidents are rare, a recent ransomware attack ended up infecting a power injector [71] responsible for injecting contrast agent for CT and MRI scans.

Industrial Control Systems (ICS): Interest in ICS security has skyrocketed in the wake of *Stuxnet* [413] with attacks being demonstrated on everything from PLCs [21], RTUs [287] and HMI [178] to DTM components [286] and rootkits [289, 393], worms [425] and ransomware [325] having been developed specifically for ICS devices. Meanwhile, recent real-world incidents targeting industrial control systems have caused blackouts on the Ukrainian power grid [178, 278, 323] and physically damaged a German steel mill [277].

Automotive: Modern automobiles are complicated networks of computer systems, essentially *"a computer with four wheels and an engine"* [203], with an ever-growing and highly vulnerable attack surface [100, 212, 330]. Attacks have been demonstrated on everything from the CAN bus and Engine Control Unit (ECU) [36, 62, 130, 301, 309–311, 379, 444], Anti-lock Braking System (ABS) [464], Tire Pressure Monitoring System (TPMS) [249, 366] or infotainment and telematics systems [308, 361, 385, 432] to Remote Keyless Entry (RKE) [251, 341] and Immobilizer [168, 169, 429, 443] systems.

Smart Homes and Consumer IoT: The rise of consumer-oriented IoT gadgets, particularly in the *smart home* sphere, is just a fragment of the total consumer electronics market but has seen a proliferation of so-called *'junk hacking'* [86, 150] targeting almost every type of device including smart home gadgets [318], alarm systems [34, 127, 468], locks and access control systems [27, 144, 191, 297, 477], surveillance cameras [44, 92, 438, 479], thermostats [349] and smart TVs [70, 400, 440].

Defense & Aerospace: 'Cybersecurity' in the context of military embedded systems has received increasingly more attention with the recognition of 'cyberspace' as a fifth domain of warfare. Given the overlap with more established military security concerns such as *Electronic Warfare (EW)* [87, 362] and *Communications Security (COMSEC)* the dividing line with 'cyber' becomes blurry. For obvious reasons, there is little detailed public research regarding attacks on military and aerospace systems but one related area that stands out for its accessibility to independent researchers is that of *Unmanned Aerial Vehicle (UAV) / drone* security. This is mainly due to the proliferation of and subsequent security research on commercial UAVs [32, 113, 196] but recent work has shown government UAVs, including those used by the military, are vulnerable as well [32, 166, 294, 382]. In addition, real-world incidents have taken place over the past decade involving signals interception [10, 114, 252] and hijacking [173] of military UAVs.

2.3 EMBEDDED PATCHING ISSUES

When a vulnerability is discovered in a computer system a patch should be created and applied, ideally as soon as possible. In order to avoid making constant patch application the responsibility of users, many general purpose operating systems contain integrated functionality for secure and automated patch management. The embedded world is very different in this regard where many systems require manually applied updates or are unpatchable altogether and vendors sometimes don't even provide a patch in the first place. There are various reasons why embedded patch management is complicated [202, 423]:

1. **Broken Patch Management**: Effective patch management requires an infrastructure that ensures a patch gets securely from the party responsible for creating patches to all affected devices with minimal interaction. For embedded systems, the problem starts with **fragmented responsibility**. As discussed in more detail in Section 6.1, no single entity manages the entire software development lifecycle of an embedded product. As a result, it is not always clear who is responsible for creating a patch when a vulnerability is discovered nor are all parties equally incentivized to create one which leads to an absence of (timely) fixes for embedded vulnerabilities. Industrial control systems are illustrative of the latter issue as shown by a 2016 *FireEye* study [66] which found that out of 1552 vulnerabilities examined over the course of 15 years, 33% had no vendor fix at the time of public disclosure. Another recent ICS vulnerability study, by Andreeva et al. [417], found that patches were available for 85% of publicly disclosed vulnerabilities, leaving the rest with either no or partial patches.

   In absence of a single, coherent patch management solution patches end up getting pushed down the supply chain on an individual or batch basis. For example, when a vulnerability is discovered in an embedded operating system, the OS vendor creates a patch and announces its availability via an update feed. It is then up to any customers such as *original equipment manufacturers (OEMs)* or *original device manufacturers (ODMs)* to monitor this feed, apply patches to their products, rebuild their firmware images and distribute them to their own customers. The further removed the party responsible for creating the patch is from the end-user of an affected product, the more complicated and slower this process becomes.

   Partially as a result of this situation, many embedded systems don't even have a **secure updating mechanism** to apply patches. While there are over-the-air software updating solutions for Linux-based embedded devices [207] and firmware updating mechanisms for specific MCUs [11, 416], these are far from universally deployable. When it comes to finally applying a patch, responsibility is not always clear either. Many embedded systems don't have a real system administrator (eg. who is responsible for patching building automation systems or smart locks?) or have the end-user/owner as a potential adversary (eg. set-top boxes) and many embedded devices vendors don't consider the products they sell to require maintenance, especially if they're disposable consumer electronics. Those vendors offering repair and replacement services, as part of a warranty or otherwise, expect to do this only in case of field defects. Some vendors try to solve this problem by offering remote maintenance services but these can become a security concern themselves as an external avenue of attack [125] or through the introduction of backdoors [83, 195, 201].

2. **Availability Issues**: Another concern is that many embedded systems are engineered for high availability. Industrial control or avionics systems, for example, have high uptime and responsiveness requirements which translate to small maintenance windows. When a system lacks hot-patching capabilities or a patch must be applied to any part of the system that requires downtime, this means patches will get applied only very infrequently.

3. **Safety Issues**: Some systems have clear risks associated with failure: eg. avionics, automotive and ICS failures can result in serious equipment damage, personal injuries or even loss of life. As a result, such *safety-critical* systems are designed and built using *safety engineering* [374] techniques such as *failure mode and effects analysis (FMEA)*, *fault tree analysis (FTA)* and *probabilistic risk assessment (PBA)*. These techniques seek to assess and minimize the risk of system failures and provide (different degrees of) reliability. Apart from careful coding, reviewing and verification some approaches include generating code from specifications using a certified production system or using formal methods to prove a piece of code meets certain requirements. Many embedded industries have also developed rigorous qualification and certification procedures such as `DO-178B` for avionics, `ISO 26262` for automotive, `IEC 62304` for medical and `IEC 61513` for nuclear.

The issue for embedded patching here arises [420, 423] from the fact that the introduction of changes in such systems requires **re-certification**. Re-certification often applies to the entire system rather than the modified components alone, especially in **mixed criticality** systems where safety- or security-critical components exist next to less critical ones, eg. infotainment systems in airplanes or cars. Since this is a costly and time-consuming process a common strategy employed to minimize re-certification cost is to delay changes until major upgrades and re-certify in bulk, but such a strategy is incompatible with the need for timely application of vulnerability patches.

The above patching issues contribute to long *vulnerability exposure windows*. The *window of exposure* of a given system to a particular vulnerability covers the entire period from the vulnerability's first discovery to the system being patched. The size of this window is determined by a) the time between *vulnerability discovery* and *vendor awareness*, b) the time between *vendor awareness* and *patch creation* and c) the time between *patch creation* and *patch application*.

Various factors play a role here such as how often the vulnerability is rediscovered (bug collision rate) and whether any of those parties notify the vendor, how often the vulnerability is exploited in the wild and whether the vendor is alerted in this manner, the length of the distribution chain between patch creator and applier, the regularity of updates, etc. On top of that embedded devices tend to have long life-spans which sometimes exceed the vendor support period and result in so-called *'forever days'* [84] which remain unpatched forever.

As a result embedded exploits tend to have a long *shelf life* (ie. remain usable against a large number of active systems) and a correspondingly high *return on investment (ROI)* for the attacker. While there are, as of writing, no studies dealing specifically with embedded exploits in this regard, research into exploit shelf life for (mainly) general purpose systems indicates this can span years. A 2012 study by Bilge et al. [386] found that zero-day attacks, ie. the period of active exploitation of a vulnerability, lasted 312 days on average. A 2017 RAND study [388] found zero-day exploits to live 6.9 years on average and have a median survival time of 5.07 years. Given the above discussed problems with embedded patch management, it seems fair to conclude that those figures are even higher for embedded exploits.

2.4 MEMORY CORRUPTION VULNERABILITIES

Binary security, the subject on which this work focuses, is a shorthand for all things related to *memory corruption vulnerabilities*. Memory corruption vulnerabilities [390, 459] are one of the oldest and most widely exploited classes of vulnerabilities in computer systems and arise from the use of so-called *unsafe languages*. As various studies have shown [88, 240–242], unsafe languages dominate the embedded world with between 60% and 66% of embedded projects written in C for the period 2011-2015 [242] compared to 71% in 2017 [88] and between 19% and 22% written in C++ for the period 2011-2015 [242] compared to 22% in 2017 [88].

And while strict programming guidelines and development practices can help reduce the number of bugs that end up in shipped code, some will inevitably slip under the radar as evidenced by memory corruption issues causing unintended acceleration problems in the Toyota Camry [14] and Audi 5000 [15] vehicles and miscalculation issues in the Patriot missile defense system [16], all with lethal consequences. So it should come as no surprise that some of the memory corruption bugs that end up in shipped embedded code are exploitable vulnerabilities. In fact, memory corruption issues consistently rank among the most prevalent vulnerability classes in embedded systems [49, 97, 98] with a 2016 Kaspersky study of ICS vulnerabilities even identifying them as the biggest single vulnerability class affecting industrial control systems [417].

2.4.1 *Language Safety*

Before we discuss memory corruption exploitation, let us first briefly get at the root of the problem by looking at *language safety*. What precisely constitutes a 'safe' language [180] (and to what degree it is safe) is a subject of some debate but generally speaking, it comes down to ensuring programs have precise and clearly defined semantics by providing *memory safety* and *type safety*. A safe language is obliged to detect or prevent unsafe behavior either at compile-time, run-time or a combination thereof and ensure that a safety violation always results in precisely defined error behavior such as throwing an exception. A language is said to be *memory safe* if and only if programs written in it are *"guaranteed to only access memory locations that they are permitted to access"* [180] and thus detect or prevent memory-unsafe behavior such as pointer arithmetic, unbounded memory access or unconstrained casting. A language is said to be *type safe* if it *"guarantee[s] that executing well-typed programs can never result in type errors"* [180]. Examples of *unsafe* languages are C(++) or assembly while examples of *safe* langues are *Ada*, *Erlang* or (to a lesser extent) *Java* or *C#*. Safe languages are not completely fail-safe, however, seeing as how their execution might rely on an interpreter or virtual machine which itself is written in an unsafe language or might have bugs in the type checker or compiler.

As mentioned earlier unsafe languages dominate the embedded world, but not exclusively. `Ada` and `Rust`, discussed among related work in Section 10.2, are two examples of safe languages with respectively a long track record and a promising future of embedded usage. However, despite the availability of such languages there are many reasons [88, 213, 420] for the continued dominance of C(++) in the embedded world:

- **Portability**: There is a C(++) toolchain for nearly every platform and operating system and many proprietary chips only come with a vendor-supplied C toolchain. If a given embedded project is to be portable across different platforms and operating systems, then C(++) tends to be the best choice.

- **Functionality**: The 'close to metal' nature of languages like C and the (safety breaking) ability to perform operations like pointer arithmetic make it well suited for the kind of optimized, low-level programming required for embedded systems.
- **Overhead**: As a result of language safety mechanisms, performance overheads resulting from run-time checks or lack of optimization due to compile-time checks might become unacceptable for a given embedded system.
- **Maturity**: C and C++ are mature languages with a long history of usage in the embedded world. As such there are many integrated development environments (IDEs), frameworks and libraries to draw upon which shortens development time.
- **Legacy Code**: There are billions of lines of legacy code written in C(++), many of which are reused in subsequent projects [242]. Porting these to a safe language (or the other way around), even if aided by automated tools, is a complicated, costly and messy affair [116] given issues with (re)verification of code, changed performance, timing and memory usage characteristics, switching to new development environments and the fact that programmers will possibly have to be re-trained in the new language. In many scenarios there simply won't be a strong business case for doing this.

2.4.2 *'Weird Machines' & Exploitation*

Some memory corruption bugs are vulnerabilities which could be leveraged by an attacker to maliciously subvert the system, for example by leaking sensitive information, escalating privileges, inducing undesirable behavior or executing arbitrary code. It should be noted that not all memory corruption bugs are vulnerabilities and not all vulnerabilities are exploitable. On the other hand, the status of a bug as vulnerability or a vulnerability as exploitable is not static either. A vulnerability occurring in one place in the program code might be unexploitable while it might be exploitable in another, and a bug that is unexploitable today might become exploitable tomorrow as a result of changes in the code base (eg. components being added or removed, control-flow paths changed, modifications to system design, etc.) and vice versa.

The exploitation of memory corruption vulnerabilities often tends to be perceived more as an arcane craft based on hazy folklore than as an exact science, a view strengthened by both the target-dependent nature of many exploits and the changes in approach as a result of the 'mitigation-bypass-counter-mitigation' cat and mouse game. But exploit development has a workflow not completely unlike that of regular software engineering [435] where the exploit developer identifies computational structures in the target that allow them to manipulate the program state on the basis of inputs and distills from these structures so-called *'exploit primitives'* (eg. memory read and write operations). The exploit developer then combines these inputs and exploit primitives into an exploit that achieves the desired manipulation of the program state.

A more formalized approach to reasoning about exploits comes in the form of the concept of so-called *'weird machines'* [53, 54, 435–437], which can be summarized as the view that *"the crafted inputs that constitute the exploit drive an input-accepting automaton already implicitly present in the target's input-handling implementation, its sets of states and transitions owing to the target's features, bugs or combinations thereof"* [437]. A less concise but perhaps more intuitive way to think about *weird machines* is to think of *'weird states'* as states falling outside of a program's *'Intended Finite State Machine (IFSM)'* [54]. Certain bugs allow the attacker to enter a *weird state* and from there potentially transition into other *weird states* by means of exploit primitives, leading to a 'state space explosion' of unintended *weird states*. Exploits can then be understood as *"programs for these 'weird machines' and serve as constructive proofs that a computation considered impossible could actually be performed by the targeted environment"* [435].

2.5 EXPLOIT MITIGATIONS

If we think of exploitation as the programming of *weird machines* through the composition of individual *exploit primitives* into a single 'exploit chain' it becomes clear, as pointed out by Hawkes [91], that exploit development can be frustrated by the introduction of *exploit mitigations*: measures which *make each link harder to forge* and/or *lengthen the chain*. In other words, by requiring more exploit primitives on part of the attacker and by making each primitive harder to construct, attacker cost are raised and certain vulnerabilities become unexploitable altogether.

In this work, we will borrow the general model for memory corruption exploitation and proposed mitigations introduced in [390] which we slightly augmented and illustrated in Figure 2. Note that this model does not aim to be exhaustive in terms of either vulnerabilities or mitigations covered but rather represent the most popular attack and mitigations techniques occurring in literature and practice as evidenced by compiler- [25, 48, 171], OS- and 3rd party [475] support. The mitigations are listed at a conceptual level and can often be implemented in different ways of varying degrees of strength.

Since exploit mitigations target exploitation techniques rather than the root cause of vulnerabilities they are not a silver bullet [54, 61, 91]. They do, however, make exploitation more complicated and time consuming (and thus costlier) which causes the number of attackers capable of exploiting a particular vulnerability to shrink [61, 381]. Mitigations with recurring 'bypass cost' in particular act as a multiplier to exploit development time since they need to be bypassed for each bug in a chain [52]. Mitigations have been shown to cause exploit developers to shift their focus to areas not covered by mitigations [388] and as such play a key role in zero day reduction trends [250].

Developing *reliable* and *stealthy* exploits is even more challenging than developing a simple *Proof-of-Concept (PoC)*. Attackers want exploits to work on all versions and configurations of a vulnerable target and ideally without making too much 'noise' in order to prevent discovery of their attack and possible loss of their zero-day. After all as the case of the `MS08-067` vulnerability [128] shows, crash log telemetry obtained via Windows Error Reporting allowed Microsoft to rapidly discover a zero-day exploit despite 95% reliability. With exploit mitigations, reliability becomes even harder to achieve as each primitive within an expanding chain will have to be reliable for the whole chain to be.

Given the above observations and the fact that more root-cause oriented solutions to memory corruption vulnerabilities in the embedded world are long-term projects at best, we consider exploit mitigations to be a viable short-term solution for increasing binary security in embedded systems. After all, as pointed out by Koopman et al. *"it might make more sense to spend a small fraction of available resources providing ways to survive bugs that will inevitably be encountered, rather than throwing all resources at an attempt to achieve absolute perfection"* and as such *"short term research milestones should emphasize characterizing practical limitations and exploring techniques to offer near-term improvement to system builders"* [423].

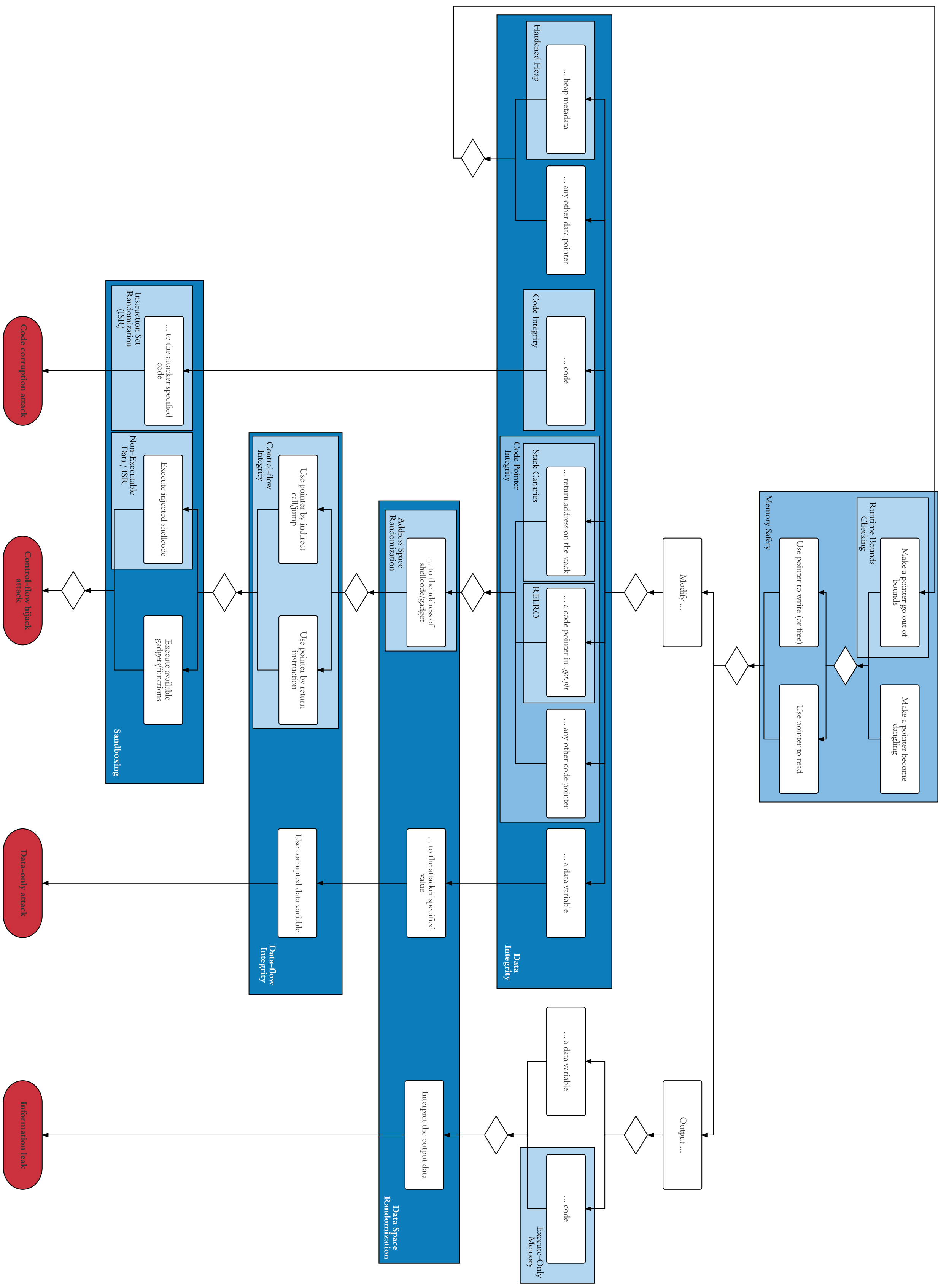

Memory Safety
Runtime Bounds Checking
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify ...
Output ...
Hardened Heap
... heap metadata
... any other data pointer
Code Integrity
... code
Code Pointer Integrity
Stack Canaries
... return address on the stack
RELRO
... a code pointer in .got.plt
... any other code pointer
... a data variable
Data Integrity
... a data variable
... code
Execute-Only Memory
Address Space Randomization
... to the address of shellcode/gadget
... to the attacker specified value
Interpret the output data
Data Space Randomization
Control-flow Integrity
Use pointer by indirect call/jump
Use pointer by return instruction
Use corrupted data variable
Data-flow Integrity
Instruction Set Randomization (ISR)
... to the attacker specified code
Non-Executable Data / ISR
Execute injected shellcode
Execute available gadgets/functions
Sandboxing
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Figure 2: Memory corruption exploitation flowchart demonstrating mitigations at different stages based on [390]

3 EMBEDDED EXPLOIT MITIGATION BASELINE

In this chapter, we will establish a minimum exploit mitigation baseline and outline their security criteria as well as their hardware and software dependencies.

3.1 ESTABLISHING A MINIMUM BASELINE

When establishing a minimum exploit mitigation baseline for embedded systems we are interested in defining an absolute minimum set of mitigations which should be reasonably expected to be present in all modern embedded systems. Because of the sheer diversity of embedded systems, we do not select our baseline on the basis of strict criteria. Instead, we only require them to be adoptable across the embedded spectrum and not rely on any specialized hardware features not commonly present in COTS embedded hardware.

Based on Figure 2, we selected the following minimum embedded exploit mitigation baseline:

1. **Executable Space Protection (ESP) / Non-Executable Data**
2. **Address Space Layout Randomization (ASLR)**
3. **Stack Canaries**

The above mitigations were selected because they are complementary and have been integrated in virtually all modern general purpose operating systems and development toolchains [48, 77, 250, 381, 388, 419], including those which are used widely in the embedded world. As such they are well understood and can reasonably be considered to be the absolute minimum in modern exploit mitigations. As noted by [355]: *"when these three techniques are properly implemented on a system they provide a strong defense against most memory error exploitations"*.

ESP forces attackers to use a *code-reuse* payload which is complicated by **ASLR**'s randomization of code and data memory. **Stack canaries** prevent stack buffer overflows from overwriting the saved function return address by requiring attacker knowledge of a secret *canary value*. **ESP** and **ASLR** apply to all control-flow hijacking attacks and while **stack canaries** only protect against stack buffer overflows, these are far more common in embedded systems than they are in modern general purpose ones [388, 417]. In addition, properly configured **ESP** (as discussed below) protects against code corruption attacks as well by making code memory non-writable. We will explain these mitigations in more depth in Section 3.2.

Baseline Limitations: Our baseline does not seek to offer exhaustive protection against all memory corruption classes and only considers control-flow hijacking and code corruption attacks while leaving data-flow and information leaks attacks unaddressed. In addition, as is usually the case with mitigations, the baseline does not guarantee absolute unexploitability of vulnerabilities but rather raises attacker cost by lengthening the exploit chain.

Keep in mind, however, that while this baseline is an absolute minimum upon which future embedded binary security work should build, embedded software has a far smaller attacker interaction surface than popular targets on general purpose machines which tends to mean that bypassing mitigations is more complicated. As pointed out by Mark Dowd [52], there have been almost no publicly known exploits achieving unconstrained code execution for Microsoft server software (eg. RPC, IIS, etc.) after the introduction of ASLR and DEP (ESP for Windows) while client software such as browsers, office suites, and PDF readers continue to be targeted by increasingly complex exploits. The key difference here is attacker control over scripting languages, multimedia elements and other interactive components which present both a larger attack surface as well as providing more granular control over the target program and memory state.

3.2 BASELINE MITIGATIONS IN-DEPTH

In this section we will discuss our mitigation baseline in-depth and outline the criteria for their successful implementation, because plenty can go wrong here which leads to easily bypassable mitigations.

3.2.1 *Executable Space Protection (ESP)*

Before we can discuss the rationale behind ESP, we first need to consider the fact that there are essentially two main processor architectural style: *Harvard* and *Von Neumann*.

Harvard and Von Neumann Architectures

The **Harvard CPU architecture** is defined as *"a processor architecture that separates data and instructions into different memory spaces. The Harvard architecture is most popular on DSPs, where the benefit of simultaneous instruction and data fetches can significantly increase signal processing throughput."* [402] The **Von Neumann CPU architecture** is defined as *"a processor structure based on a basic theory of operation that intermixes data and instructions so that any value in memory can be executed or interpreted as data. With the exception of some supercomputers and most DSPs, the von Neumann architecture is predominant"* [402].

Since in this work we are only concerned with how these architectural differences relate to memory corruption exploitation, we define a processor to have a Harvard architecture *iff* it has separate code and data memories (from the programmer's external point of view) and its data memory is non-executable. This somewhat relaxed definition allows code memory to be generally readable and sometimes writable too under limited circumstances (eg. limited to privileged instructions executed by a bootloader). All architectures which do not meet our Harvard definition are considered Von Neumann.

This architectural distinction is important when considering that an attacker executing a control-flow hijacking attack seeks to redirect intended control-flow to a piece of code of their own choosing. On a Von Neumann processor, the attacker can inject any piece of code into data memory and redirect control-flow to it to achieve arbitrary code execution. However, on a Harvard processor the attacker is restricted to repurposing fragments of pre-existing code memory by means of a so-called *code-reuse* attack.

Code-Reuse Attacks

Code-Reuse Attacks [458] are a class of control-flow hijacking attacks where an attacker re-purposes existing code to a malicious end, thus effectively constructing a malicious payload out of pre-existing program code without having to inject a malicious payload into memory. Code-reuse attacks evolved out of the more specific `Return-Into Libc (RILC)` [405] attacks where whole functions are re-purposed and encompasses variants where fragments of code (often referred to as *gadgets*) are strung together in 'chains' such as `Return-Oriented Programming (ROP)` [279, 430], `Jump-Oriented Programming (JOP)` [458], `Call-Oriented Programming (COP)` [410], `Sigreturn-Oriented Programming (SROP)` [336] and `Counterfeit Object-Oriented Programming (COOP)` [339].

ROP is by far the most common and popular code-reuse attack class and there are various interactive and automated tools [111, 123, 199, 332, 469] to aid in constructing ROP chains. While ROP attacks have been shown to be Turing-complete on several platforms [337, 454], practical limitations [292] such as available gadgets, available memory or payload size restrictions mean ROP chains are often far less expressive in practice and many attackers tend to prefer 'partial ROP' [31, 279, 471] attacks where an initial ROP payload is used to change memory permissions and circumvent ESP thus allowing for injection of a second stage payload containing the actual malicious code.

Executable Space Protection (ESP) (also known as DEP, NX or $W\hat{X}$ memory) essentially seeks to emulate Harvard-style code and data separation on a Von Neumann processor by rendering data memory non-executable as well as usually ensuring code memory is non-writable. The goal here is to force attackers to use code-reuse attacks which are both more complicated to execute as well as often more restricted in capabilities.

ESP can be implemented either by relying on hardware support or by means of software emulation. The former case is usually implemented in the form of a dedicated hardware feature, usually part of the *Memory Management Unit (MMU)*, regulating memory executability permissions at a certain granularity level such as on a per-page basis.

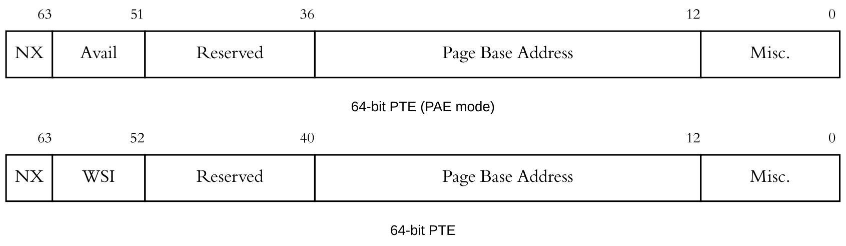


Figure 3: x86 Page Table Entry (PTE) with NX bit

Hardware ESP

We define **hardware ESP** support as support by a Von Neumann CPU for a feature (usually as part of integrated MPU or MMU functionality) that allows for marking memory as (non-)executable. This does not have to have been desigend or marketed explicitly as an 'ESP' feature by processor vendors for it to serve this purpose. Examples of hardware ESP features are AMD's `NX` bit (illustrated in Figure 3), Intel's `XD` bit, ARM's `XN` bit, MIPS' `XI` bit and the PowerPC segment register `N` bit or TLB entry `EX` bit.

There are multiple approaches to software-based ESP, the most well-known being the PaX project's implementation [174], the details of which are out of scope for this work. All software emulation approaches, however, incur at least some runtime overhead and tend to be architecture-specific in design.

Regardless of whether the design is hardware- or software-based, implementing ESP has an operating system component and a toolchain one. Care should be taken to meet the following criteria:

1. **Default Policy**: The OS components implementing ESP should have a default policy marking data memory (stack, heap, etc.) as non-executable and code memory as non-writable by default. In cases where backwards compatiblity with ESP-violating constructs (eg. dynamically generated or self-modifying code) is required, ESP policy should be enforced on an *opt-out* basis to avoid putting the burden of proper configuration on system integrators and administrators as is the case with *opt-in* policies [471].

2. **OS Support**: The OS memory management subsystem and program loader should ensure memory areas (eg. code memory for a loaded program, stack, and heap, etc.) are marked as per their default policy upon allocation.

3. **Toolchain Support**: The toolchain should ensure proper separation of code and data is respected by avoiding constructs which violate this (such as code trampolines, data inlined in code, etc.).

   If *opt-out* support is required the toolchain should emit a marker indicating this (such as the `GNU_STACK` ELF header [20]), either indicating explicit memory permissions for a given memory element overriding ESP policy or a generic *opt-out* causing a fallback to non-ESP defaults.

4. **Coverage**: Shared libraries loaded by a program should fall under the program's ESP policy to prevent *'split personalities'* [208] from arising in case of *opt-out* marking conflicts.

5. **Granularity**: Care should be taken to consider the granularity level at which memory permissions are applied on a given platform to prevent ESP violations caused by overlapping or incompletely covered memory regions arising from improper alignments or data and code separation boundaries.

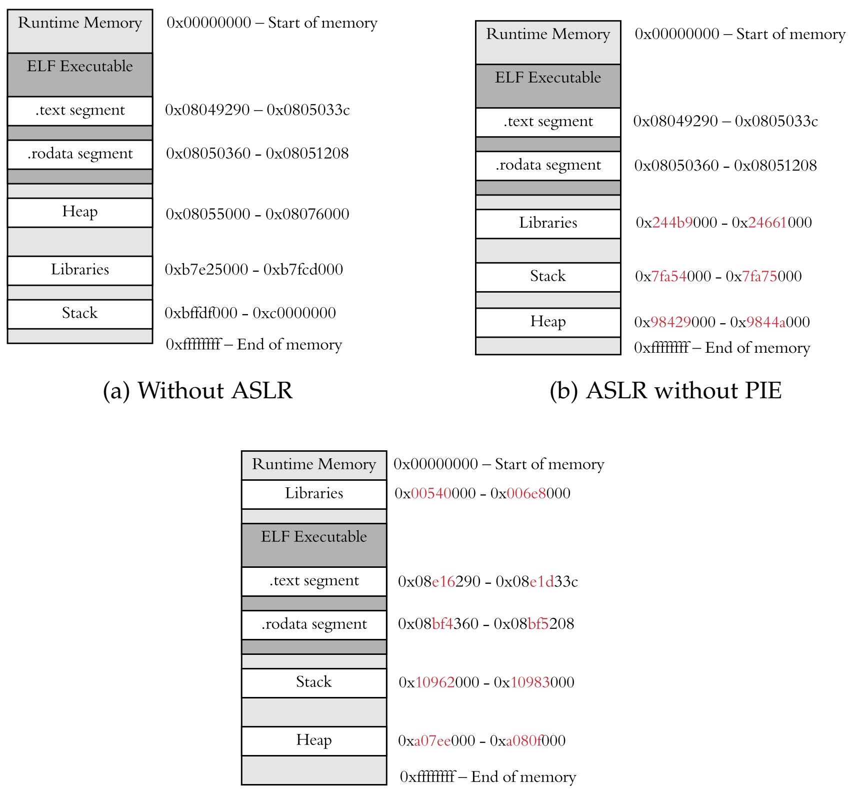
6. **Conflicting Features**: Any features which conflict with ESP policies, such as the Linux `READ_IMPLIES_EXEC` [51, 95, 152] 'legacy support' personality flag, should be turned off by default.

3.2.2 *Address Space Layout Randomization (ASLR)*

When developing exploits, attackers rely on knowledge of the target application's memory map for write and read targets as well as crafting code-reuse payloads. **Address Space Layout Randomization (ASLR)** [22, 81] is a technique which seeks to break this assumption by ensuring memory layout secrecy via randomization of addresses belonging to various memory objects (eg. code, stack, heap, etc.) and rendering them hard to guess as illustrated in Figure 4. A full overview of all existing ASLR designs and proposals is out of scope for this work but they can be categorized based on the following decisions:

1. *When to (re)randomize*: Randomization can occur at different points in time: at boot, when a program is loaded, when a thread is created, etc. This choice determines the variation in memory layout between different processes running on the same machine and the options for *relocation frequency*.

2. *What to randomize*: Program memory can be seen as divided into different memory objects all of which can be randomized to different degrees. In order to thwart code-reuse attacks, at least code memory (both that of the main program image and any



(a) Without ASLR

(b) ASLR without PIE

(c) ASLR with PIE

Figure 4: Address Space Layout Randomization (ASLR) [22]

shared libraries) needs to be randomized. Depending on the operating system design, randomization of certain memory objects might require them to be compiled in a special fashion such as *Position Independent Executables (PIE)* required for main program code randomization as illustrated in Figure 4b versus Figure 4c.

3. *How to randomize*: Randomization can occur at different levels of granularity (eg. only randomizing memory object base addresses vs. randomizing their internal layout) and with different degrees of correlation (eg. loading all shared libraries in-order at static offsets from the first or randomizing them individually).

It is crucial for the effectiveness of ASLR that sensitive addresses are both *secret* and *unpredictable*. Generally speaking the more granular and frequent the (re)randomization, the better.

Over the years, ASLR implementations in various operating systems have been found to suffer from a variety of weaknesses [81, 154, 359]. Based on prior work [81, 355, 356] we can conclude ASLR security is based upon the following criteria:

1. **Entropic Quality**: The probability that an attacker can guess the locations of randomized memory objects within a reasonable amount of time needs to be low for ASLR to be secure. As such we require randomized addresses to have high entropic quality, something which has three different dimensions [355]:

    a) **Non-Randomized Sections**: A single non-randomized (code) section can often be used by an attacker to bypass ASLR if sufficient gadgets reside within it. On Linux `VDSO` (named `linux-gate.so` on some systems) [77] used to be mapped at a fixed address while Windows used to be plagued by commonly loaded old non-ASLR libraries such as `msvcr71.dll` [472] or structures mapped at fixed addresses such as `SharedUserData` or `Wow64SharedInformation` [271]. In some cases, randomized memory objects might end up being mapped at fixed addresses as an unforseen side-effect of memory pressure [60]. ASLR as implemented in popular general purpose operating systems has evolved to avoid these pitfalls by carefully randomizing *all* memory objects.

    b) **Range of Entropy**: The range of possible values for a given randomized address determines the probability that an attacker can guess the address within a reasonable amount of time. this range is limited by the available (virtual) memory space and the flexibility of memory object placement the operating system allows.

    c) **Relocation Frequency**: Address randomization can occur at different points in time and either once or more frequently. Ideally, all memory objects for different processes are all mapped at different locations with respect to both previous execution runs and other processes. It is also possible to relocate memory objects during runtime so that addresses are not only randomized per execution run and per process but also randomized throughout a single run. While the latter approach is not integrated into the ASLR implementation of any major operating system and comes with serious overhead, it has the additional benefit of limiting the usefulness of information leaks.

2. **Brute-Force Resistance**: In addition to the entropic quality, ASLR designers need to take *memory layout inheritance* into account when seeking to harden against brute-force attacks. Many operating systems have process management models where child processes inherit the memory layout of their parents. This has two major drawbacks: a) It makes application architectures where children are respawned after crashing highly vulnerable to brute-force attacks b) Child processes know the memory layouts of sibling processes which constitutes a breach of ASLR's memory

secrecy assumption if child processes are considered independent (eg. Android's `Zygote` sandboxing model).

The former is particularly dangerous in the light of *byte-by-byte* brute-force attacks against ASLR [273, 356]. Here an attacker with fine-grained control over their memory corruption vector targeting an application with respawning child processes or threads starts by overwriting the first byte of a given codepointer and observes the resulting control-flow. If the target thread or child process crashes their guess was incorrect, while if it continues running as intended it was correct. Because of *memory layout inheritance*, the codepointer value does not change between guesses and the attacker can try all 256 possible values before obtaining the correct value for the first byte and move on to the next one. In this manner, the attacker only needs at most 256^N trials to brute-force the target codepointer, where N is the codepointer byte-width.

3. **Information Leaks**: Since ASLR security rests on memory layout secrecy, *information leaks* [206] disclosing memory layout or contents (eg. in the form of code or data pointer values) can be used to defeat it. The impact of a particular information leak depends on the scope and flexibility of the leak itself as well as the ASLR implementation but generally speaking they are the most reliable way to defeat modern ASLR implementations. While information leaks might result from and require an attacker to find an additional bug on top of the one used to hijack control-flow, sufficiently flexible vulnerabilities can often be crafted into information leaks themselves. In addition, operating system designers need to consider memory layout and content as secret when introducing ASLR and ensure they are not disclosed to potential attackers by system features [238, 452].

4. **Correlation Attacks**: When the location of one memory object can be used to deduce the location of others these are said to be correlated. Correlation can either be *total*, where the attacker can obtain the exact location of a correlated memory object, or *partial*, where the attacker's search space is reduced by using correlation information. Many popular ASLR implementations suffer(ed) from some form of correlation, allowing attackers to leverage less powerful or even otherwise useless information leaks to fully defeat ASLR. An example of a *total* correlation attack is the `offset2lib` [355] attack affecting PIE binaries on Linux and the ASLR bypass used in the `Metaphor` exploit [19] for the `Stagefright` vulnerability.

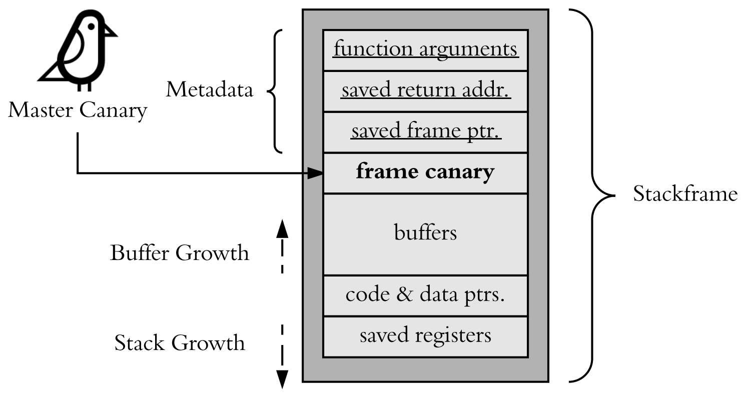


Figure 5: Stack Canary Example

3.2.3 *Stack Canaries*

A **Stack Canary Scheme** [190] is a security mechanism that aims to protect against 'linear' stack buffer overflows. That is, stack buffer overflows that seek to overflow a local buffer into local stackframe metadata such as the saved frame pointer or return address. It does this by placing a so-called *canary* or *guard* value between the metadata and local variables. Over the years, stack canaries have evolved and have seen many incarnations in popular toolchains and operating systems [137, 190, 470] but they are generally impelemented as a compiler extension (with an operating system component) which:

1. Inserts code to generate a random *master canary value* at program startup and store it somewhere, preferably in a location that cannot be overwritten or read without invocation of a dedicated instruction (eg. in a special data segment).
2. Instruments function prologs to push a copy of the master canary value to the stack between stackframe metadata and local variables, as illustrated in Figure 5.
3. Instruments function epilogs to pop the saved canary value from the stack and compare it against the master canary. If they don't match, the stack has been corrupted and a violation handler is invoked (usually terminating the application) in order to raise an alert and prevent control-flow hijacking from occurring.

By drawing upon the history of stack canary schemes [77, 151, 190, 194, 273, 419, 470] and prior work [81, 354] we can outline the following security criteria:

1. **Protection Coverage**: The canary only protects the elements it shields from local variables. Thus all stackframe metadata, and

ideally any stack-stored function arguments as well, should be placed behind the canary.

2. **Data and Pointer Separation**: Since the canary does not cover local variables, an attacker can still overwrite those and potentially hijack control- or data-flow by targeting any local code- or data-pointers. The likelihood of this being feasible can be reduced if the compiler reorders local variables so that non-buffer data (especially any code pointers) are located below the buffer.

3. **Function Coverage**: Since canary protection works on a per-stackframe basis, every function in a given application has to be protected individually. Since this can come with code size and performance overhead that might be unacceptable, many canary schemes allow for configurable function coverage (eg. based on a minimum buffer size) such as GCC's `-fstack-protector*` flags.

4. **Canary Type**: Over the years different types of canaries have been proposed to complicate stack buffer overflow exploitation. The main types are:

   a) **Random Canary**: Generated using a secure random number generator. Because the value is random the attacker cannot predict it and does not know the correct value with which to overwrite the stored canary value.

   b) **Terminator Canary**: These canaries are set to a combination of NULL, `CR`, `LF` and `0xFF` values which act as terminators for various types of parsing and data handling functions (eg. `strcpy`) in order to terminate an overwriting operation prematurely. All-`NULL` canaries can be considered a sub-type of terminators.

   c) **XOR Canary**: Consist of a combination of a random canary value and stored control data such as the saved return address.

   Generally, the preferred canary type is a mixture of random and terminator canaries.

5. **Canary Secrecy and Entropy**: Canaries which include a random component have two important security properties, they should be: a) Secret and b) Unpredictable. Apart from information leaks, the main threat to canaries of this type is low entropy rendering the canary predictable. In order to prevent the latter, canaries should be 'wide' enough to accommodate a sufficient number of random bits and these bits should be generated using a *cryptographically secure random number generator (CSPRNG)*. As the impact of low boot entropy of the Linux CSPRNG on Android on its stack canary implementation has shown [326, 327,

465], a proper evaluation of stack canary scheme security thus also requires an evaluation of the utilized CSPRNG.

6. **Susceptibility to Brute-Force**: Since randomization-style canaries rely on an attacker being unable to guess the canary value, they need to be hardened against brute-force attacks. Sufficient canary entropy is one aspect of this but, as with ASLR, operating system design details should be taken into consideration as well. In most canary schemes the master canary value is generated once and shared across forking and threading operations.

   In application architectures where child processes and threads are respawned upon crashing, this renders the canary scheme particularly vulnerable to brute-force attacks because they can try to exploit the vulnerability with new canary guesses upon each crash. *Byte-for-byte* brute-force attacks on the canary [273, 354] are particularly dangerous here because this allows an attacker to brute-force the canary with at most 256^N guesses where N is the canary byte-width minus the number of terminator-style bytes. Thwarting these attacks would require a canary scheme incorporating canary renewal [354].

3.3 EXPLOIT MITIGATION DEPENDENCIES

The exploit mitigations in our baseline have a variety of software and hardware dependencies that have to be supported by a particular system in order for a mitigation to be deployable on it. In this section, we map out and discuss these dependencies for each of our baseline mitigations.

3.3.1 *ESP Dependencies*

Figure 6 illustrates ESP's dependencies. All ESP implementations require some form of memory protection support on part of the operating system, either in the form of simple permissions or in the form of more elaborate protection management.

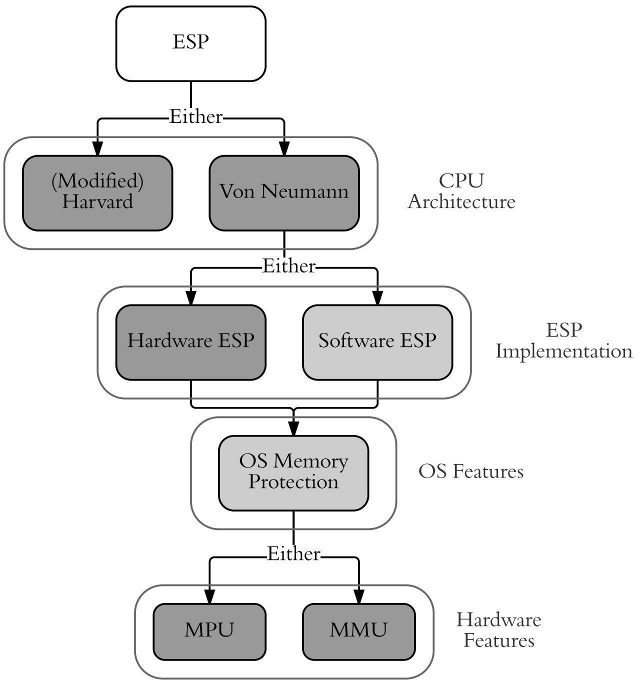


Figure 6: ESP Dependencies

Memory Protection

Memory Protection can be defined as access control management for memory. Its main goal is to prevent software faults by disallowing (a certain type of) access to a given memory region to code that does not have the proper permissions. This can be as trivial as setting read and write permissions at various levels of granularity but can also involve a more complex full-protection model with private virtual memory per process. There are many different approaches to memory management and protection but for our purposes, we are only interested in an operating system's capabilities to manage the most basic form of memory protection: managing permissions (eg. read, write, execute) at a certain level of memory granularity. In this work, we will consider an OS to have memory protection capabilities if it is at least able to manage permissions.

In order for an operating system to provide memory protection support, it requires the system to have a Memory Protection Unit (MPU) or Memory Management Unit (MMU). While there are external MMUs available for some processors, this is not the case for the vast majority of them and as such many low-end microcontrollers are excluded from providing memory protection support.

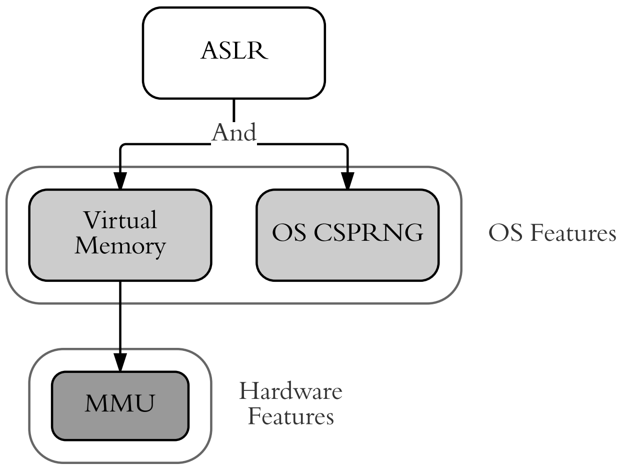


Figure 7: ASLR Dependencies

MPUs and MMUs

A **Memory Management Unit (MMU)** is *"a quite complex circuit that translates logical addresses to physical addresses. An MMU is used to manage the memory needs of multiple processes in a single physical memory. The MMU segments physical memory into many frames, each of which has its own (read/write/execute) access rights. A particular process can run only within its own frames; any attempt to access other frames results in a fault. The OS captures the fault and takes appropriate action."* [402] The MMU is typically integrated as part of the CPU in modern processors. A more lightweight variant, the **Memory Protection Unit (MPU)**, provides support for memory protection capabilities but not virtual memory management. We consider this the prime distinguishing feature between MPUs and MMUs.

3.3.2 *ASLR Dependencies*

As illustrated in Figure 7, ASLR, first of all requires *virtual memory* support on part of the operating system. This is necessary because when ASLR places memory objects belonging to a given process at a randomized location, it needs to ensure no other memory objects belonging to any other process have already been placed there in order to avoid shared memory conflicts. Virtual memory allows for per-process private memory which prevents shared memory conflicts by isolating processes from each other.

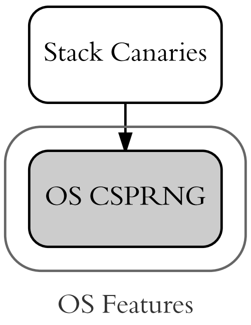


Figure 8: Stack Canary Dependencies

Virtual Memory

Virtual Memory is *"a scheme that permits a system to use vast amounts of memory, even exceeding the amount that physically exists on the system. Virtual memory systems use a paging unit, which (like an MMU) translates addresses into physical addresses. The paging unit, though, also tracks the location of a page of virtual memory, which can be in physical memory or off on a hard disk. If not in memory, it's swapped in from the disk."* [402]

In order for the operating system to offer virtual memory support, it needs an MMU to facilitate address translation. Finally, ASLR requires an operating system CSPRNG in order to generate strongly random numbers for address randomization purposes.

OS Cryptographically Secure Pseudorandom Number Generator (CSPRNG)

We define an **OS CSPRNG** to be a cryptographically secure random number generator provided and managed by the operating system such as */dev/random* on Unix-like systems or the *CryptGenRandom* [143] API on Windows systems. This includes entropy source management and reseed control.

3.3.3 *Stack Canary Dependencies*

As illustrated in Figure 8, the only stack canary dependency is OS CSPRNG support in order to generate unpredictable canary values.

Part II

ANALYSIS OF EMBEDDED EXPLOIT MITIGATIONS

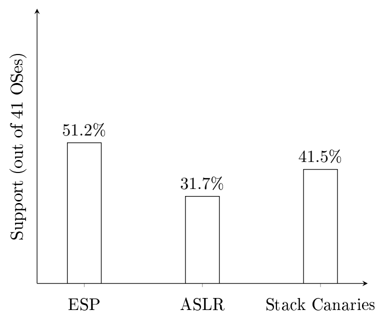
4 QUANTITATIVE ANALYSIS

In this chapter we will present the, to the best of our knowledge, first quantitative evaluation of exploit mitigation adoption (as per our baseline outlined in Section 3.1) and dependency support (as outlined in Section 3.3) among popular embedded operating systems and hardware. While the results of our quantitative evaluation are not weighed by market share and hence are not a direct reflection of the current embedded systems market, they are a reflection of the current embedded *state of the art* allowing us to identify clear gap areas as well as core challenges (outlined in more detail in Section 6) faced by embedded operating system developers adversely affecting mitigation adoption.

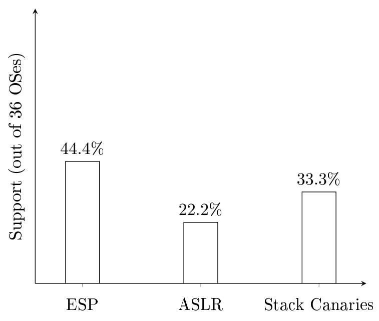
4.1 EMBEDDED OS MITIGATION & DEPENDENCY SUPPORT

Unlike the general-purpose world, which is dominated by a relatively small handful of operating systems with roughly similar capabilities, the embedded world features a polyculture of different operating systems. In order to present an overview of the current state of embedded OS mitigation adoption, we evaluated 41 popular embedded operating systems. Our selection aims to be a representative sample of embedded operating systems and includes those listed by recent UBM Embedded Markets Studies [240–242], those listed by various studies into embedded operating systems [7, 188, 269, 291, 415] as well as some of the most popular mobile operating systems [165].

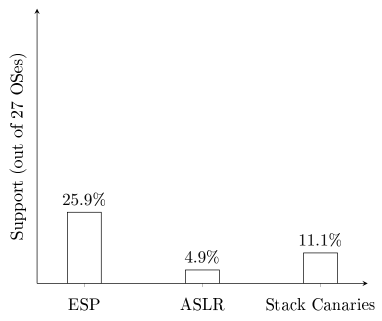
We evaluated our OS selection for exploit mitigation and dependency support through a combination of vendor surveys, documentation consultation, and experimental validation. Detailed results are reported in Tables 1 and 2 and aggregate results in Figures 9 and 10. We consider a mitigation or feature supported ***iff*** it is supported by the OS for at least some (but not necessarily all) platforms. As this is a quantitative assessment it does not evaluate the quality of the implementation nor whether the feature is enabled by default and as such the assessment is an optimistic one. In addition, we mark an operating system as providing an OS CSPRNG ***iff*** provided PRNG functionality is advertised as such or can be reasonably assumed to provide secure random number generation functionality (eg. the `/dev/random` interface on Unix-like systems). Operating systems marked in our results as lacking an OS CSPRNG might still provide regular PRNG functionality or provide support for interacting with hardware RNGs



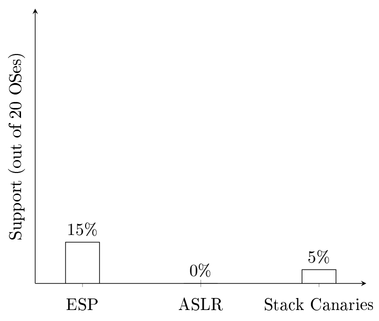
(a) All OSes



(b) Non-Mobile OSes



(c) Non-Linux/Windows/BSD-based OSes



(d) Deeply Embedded OSes

Figure 9: Embedded OS Exploit Mitigation Support

and those marked as providing an OS CSPRNG do not necessarily provide a secure one as the results of our qualitative analysis in Chapter 5 show.

4.2 EMBEDDED HARDWARE FEATURE SUPPORT

The embedded world features a wide range of different processor architectures and *'core families'* with different capabilities. In order to establish an overview of common embedded hardware capabilities with respect to exploit mitigation dependencies, we make sample selection of several *core families* and map out their architectural style and MPU, MMU and hardware ESP support capabilities.

OS	ESP	ASLR	Canaries
Android	✓	✓	✓
iOS	✓	✓	✓
Windows 10 Mobile	✓	✓	✓
BlackBerry OS	✓	✓	✓
Sailfish OS	✓	✓	✓
Ubuntu Core	✓	✓	✓
Brillo	✓	✓	✓
QNX	✓	✓	✓
`RedactedOS`	×	×	×
VxWorks	✓	×	×
INTEGRITY	✓	×	×
Yocto Project Linux	✓	✓	✓
Windows Embedded / IoT	✓	✓	✓
OpenWRT	✓	✓	✓
μClinux	✓	×	✓
CentOS	✓	✓	✓
NetBSD	✓	✓	✓
Junos OS	✓	×	✓
ScreenOS	×	×	×
Cisco IOS	×	×	×
eCos	×	×	×
Zephyr	×	×	✓
IntervalZero RTX	✓	×	✓
Enea OSE	×	×	×
ThreadX	×	×	×
Nucleus	×	×	×
NXP MQX	×	×	×
Kadak AMX	×	×	×
Keil RTX	×	×	×
RTEMS	×	×	×
freeRTOS	×	×	×
Micrium μC/OS[1]	✓	×	×
TI-RTOS	×	×	×
DSP/BIOS	×	×	×
TinyOS	×	×	×
LiteOS	×	×	×
RIOT	✓	×	×
ARM mbed	✓	×	×
Contiki	×	×	×
Nano-RK	×	×	×
Mantis	×	×	×

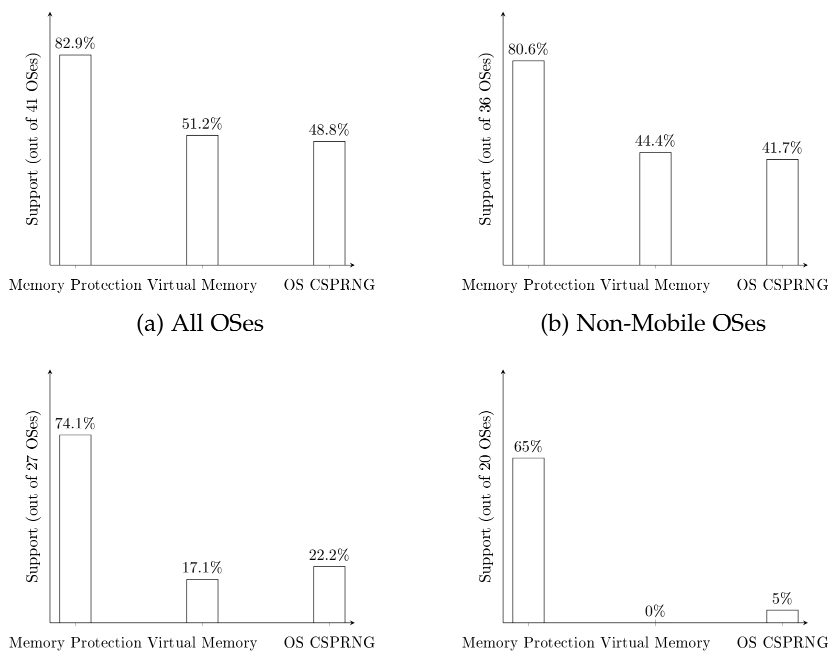
Table 1: Embedded OS exploit mitigation adoption

[1] *A μC/OS-II kernel version with ESP support is available via a Micrium partner*

OS	MPROT	VMEM	RNG
Android	✓	✓	✓
iOS	✓	✓	✓
Windows 10 Mobile	✓	✓	✓
BlackBerry OS	✓	✓	✓
Sailfish OS	✓	✓	✓
Ubuntu Core	✓	✓	✓
Brillo	✓	✓	✓
QNX	✓	✓	✓
`RedactedOS`	✓	✓	✓
VxWorks	✓	✓	×
INTEGRITY	✓	✓	✓
Yocto Project Linux	✓	✓	✓
Windows Embedded / IoT	✓	✓	✓
OpenWRT	✓	✓	✓
μClinux	✓	✓	✓
CentOS	✓	✓	✓
NetBSD	✓	✓	✓
Junos OS	✓	✓	✓
ScreenOS	✓	✓	✓
Cisco IOS	×	×	✓
eCos	×	×	×
Zephyr[1]	✓	×	×
IntervalZero RTX	✓	✓	✓
Enea OSE	✓	✓	×
ThreadX	✓	×	×
Nucleus	✓	×	×
NXP MQX	✓	×	×
Kadak AMX	×	×	×
Keil RTX	✓	×	×
RTEMS	×	×	×
freeRTOS	✓	×	×
Micrium μC/OS	✓	×	×
TI-RTOS	✓	×	×
DSP/BIOS	✓	×	×
TinyOS	✓	×	×
LiteOS	✓	×	×
RIOT	✓	×	×
ARM mbed	✓	×	×
Contiki	×	×	×
Nano-RK	×	×	×
Mantis	×	×	×

Table 2: Embedded OS exploit mitigation dependency support

[1] *While memory protection support is not yet available in the latest Zephyr release as of writing, it will be rolled out in the upcoming Zephyr 1.8 release [163]*



(a) All OSes

(b) Non-Mobile OSes

(c) Non-Linux/Windows/BSD-based OSes

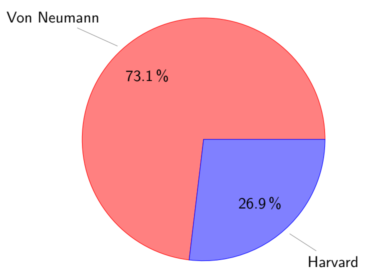
(d) Deeply Embedded OSes

Figure 10: Embedded OS exploit mitigation dependency support

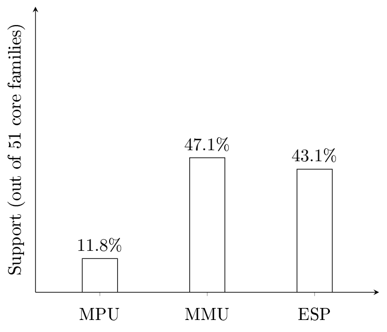
Core Family

When we talk about *'cores'* we refer a specific implementation of a (version of) a particular processor architecture. In our terminology a core (derived from ARM Holdings' use of their IP core licensing terminology) can be an individual MCU, CPU or SoC design but always refers to a specific implementation. We use this concept (in a flexible fashion) to create a single terminology and group individual cores into a *'core family'* (such as ARM's Cortex-A or Atmel's ATmega series) whose members are functionally mostly similar but might differ with respect to some features and peripherals. We base our groupings of core families on vendors' own series groupings.

We evaluated 78 different *core families* for hardware dependency support, with detailed results reported in Tables 3, 4 and 5 and aggregate results in Figure 11, through documentation and datasheet consultation. We consider a feature supported ***iff*** it is supported by all members of a given core family and absent ***iff*** it is not supported by any of them. Any variation with regards to dependency support is denoted with ~ and omitted from aggregate results. The *core families* in our selection belong to the following embedded architectures: ARM, MIPS32, PIC, PPC, x86, SuperH, AVR, AVR32, Intel 8051, Motorola 68000, Infineon TriCore, MSP430, C166, Blackfin, ARC, RL78



(a) Core Family Architectures



(b) Von Neumann Core Family Feature Support

Figure 11: Core Family dependency support

and RX. Our selection aims to be a representative sample of *core families* belonging to major architectures and vendors in the embedded space across industry verticals and includes, among others, the most popular *core families* listed by recent UBM Embedded Markets Studies [240–242] and EDN reader surveys [59].

4.3 CONCLUSIONS

Among the embedded operating systems surveyed, we can distinguish two major clusters in terms of capabilities and purposes:

- **High-End**: These operating systems are aimed at the higher end of the embedded spectrum and offer *virtual memory* capabilities as well as often being POSIX-compliant. This includes mobile operating systems (eg. Android, iOS, etc.), lightweight versions of operating systems common in the general-purpose world (such as Linux, Windows or BSD) as well as operating systems like QNX or `RedactedOS` .
- **Low-End**: These operating systems are aimed at *deeply embedded systems*, often have real-time capabilities and do not offer *virtual memory* support. As such, there is usually no separation between user- and kernelspace and instead of isolated processes, there tends to be just a kernel running a limited set of tasks. Examples are RTOSes such as ThreadX, RTEMS, Micrium µC/OS and TinyOS.

From the results in Figures 9a, 9b, 9c, 9d, 10a, 10b, 10c and 10d we can observe that all mobile operating systems have support for every exploit mitigation in our baseline and so do most Linux, BSD, and Windows-based operating systems. Outside of those, apart from QNX, almost all other operating systems lack support for any mitigations whatsoever. We can also see that while memory protection support is almost universally present, virtual memory and OS CSPRNG

Core Family	Arch.	MPU	MMU	ESP
ARM				
ARM1	N	×	×	×
ARM2	N	×	×	×
ARM3	N	×	×	×
ARM6	N	×	×	×
ARM7	N	×	×	×
ARM7T	N	~	~	×
ARM7EJ	N	×	×	×
ARM8	N	×	✓	×
ARM9T	N	~	~	×
ARM9E	N	~	~	×
ARM10E	N	×	✓	×
ARM11	N	~	~	✓
ARM Cortex-A	N	×	✓	✓
ARM Cortex-R	N	✓	×	✓
ARM Cortex-M	N	~	×	✓
PIC				
PIC10	H	×	×	×
PIC12	H	×	×	×
PIC16	H	×	×	×
PIC18	H	×	×	×
PIC24	H	×	×	×
dsPIC	H	×	×	×
MIPS32				
PIC32MX	N	×	×	×
PIC32MZ EC	N	×	✓	×
PIC32MZ EF	N	×	✓	✓
PIC32MM	N	×	✓	✓
PowerPC				
PPC e200	N	~	~	✓
PPC e300	N	×	✓	✓
PPC e500	N	×	✓	✓
PPC e600	N	×	✓	✓
PPC 403	N	×	×	×
PPC 401	N	×	×	×
PPC 405	N	×	✓	✓
PPC 440	N	×	✓	✓
PPC 740	N	×	✓	✓
PPC 750	N	×	✓	✓
PPC 603	N	×	✓	✓
PPC 604	N	×	✓	✓
PPC 7400	N	×	✓	✓

Table 3: Core Family dependency support

Core Family	**Arch.**	**MPU**	**MMU**	**ESP**
x86				
Intel Atom Z34XX	N	×	✓	✓
Intel Quark X10XX	N	×	✓	✓
Intel Quark μC[1]	N	×	✓	✓
SuperH				
SH-1	N	×	×	×
SH-2	N	×	×	×
SH-3	N	×	✓	×
SH-4	N	×	✓	×
AVR				
ATtiny	H	×	×	×
ATmega	H	×	×	×
ATxmega	H	×	×	×
AVR32				
AVR32A	N	✓	×	×
AVR32B	N	×	✓	×
8051				
Intel MCS-51	H	×	×	×
Infineon XC88X-I	H	×	×	×
Infineon XC88X-A	H	×	×	×
m68k				
NXP M683XX	N	×	×	×
NXP ColdFire V1	N	×	×	×
NXP ColdFire V2	N	×	×	×
NXP ColdFire V3	N	×	×	×
NXP ColdFire V4	N	×	✓	×
NXP ColdFire V5	N	×	✓	×
TriCore				
Infineon TC11xx	H	×	~	×
Infineon AUDO Future	H	×	×	×
C166				
Infineon XE166	N	✓	×	✓
Infineon XC2200	N	✓	×	✓
MSP430				
MSP430x1xx	N	×	×	×
MSP430x2xx	N	×	×	×
MSP430x3xx	N	×	×	×
MSP430x4xx	N	×	×	×
MSP430x5xx	N	×	×	×
MSP430x6xx	N	×	×	×
MSP430FRxx	N	✓	×	×

Table 4: Core Family dependency support

[1] *Intel Quark Microcontrollers (D1000/C1000/D2000)*

Core Family	Arch.	MPU	MMU	ESP
Blackfin				
Analog Blackfin[1]	N	✓	×	×
ARC				
Synopsys ARC EM	H	~	×	×
Synopsys ARC 600	H	~	×	×
Synopsys ARC 700	H	×	~	×
RL78				
Renesas RL78/G1x	H	×	×	×
Renesas RL78/L1x	H	×	×	×
RX				
Renesas RX200	H	~	×	×
Renesas RX600	H	~	×	×

Table 5: Core Family dependency support
[1] *Although documentation mentions an MMU it does not support address translation (and thus does not allow for virtual memory) which is why we consider it an MPU for our purposes*

support is almost universally lacking in the *low-end* operating systems aimed at deeply embedded systems. From these observations we can conclude that exploit mitigation adoption (and underlying dependency support) is generally present only on the *high-end* embedded operating systems which derive from Linux, BSD or Windows.

When it comes to the hardware *core families* surveyed, we can see from Figure 11b that under half of the (Von Neumann) *core families* in our selection have MMU support. A small minority of *core families* has MPU support, leaving just under half of the Von Neumann *core families* in our selection without necessary hardware support for memory protection and over half without the hardware support required for virtual memory. This lack of MPU and MMU support makes sense for the more constrained end of the spectrum such as MCUs which only have support for integrated memory and no support for external memory. Similarly under half of Von Neumann *core families* in the selection have hardware ESP support meaning ESP can only be implemented via software emulation on these systems. As observed by Michael Barr [13], various UBM Embedded Markets studies [240–242] and other observers [37], the embedded world is seeing a trend towards deployment of 32-bit CPUs (and for the most high-end embedded systems even 64-bit CPUs [209]) over the traditionally used 8- or 16-bit CPUs. Since most popular 32-bit architectures are Von Neumann this has security implications, though these are possibly offset by the fact that certain modern CPU architectures offer hardware ESP support (eg. ARMv6+, MIPS32r3+, x86, etc.).

Based on the above observations we can conclude that there is a clear gap area when it comes to *deeply embedded systems*. Only among the *high-end* Linux, BSD and Windows-based operating systems is there any significant exploit mitigation adoption and when it comes to *low-end* OS capabilities the lack of virtual memory and OS CSPRNG support present obstacles to ASLR and stack canary adoption.

5 QUALITATIVE ANALYSIS

In this chapter, we will present the, to the best of our knowledge, first qualitative evaluation the exploit mitigations (and OS CSPRNGs) of three embedded operating systems: *QNX* [24], `RedactedOS` and *Zephyr* [186]. We believe this selection to be representative of popular non Linux-, BSD- or Windows-based embedded operating systems.

We chose to evaluate OS CSPRNGs in addition to the exploit mitigations themselves because the security of the former is crucial to the wider security ecosystem (eg. through their usage in cryptographic software such as *OpenSSL*). In our analyses of these OS CSPRNGs we did not perform a full FIPS or NIST standards validation but instead opted for an offense-oriented approach by uncovering weaknesses using industry standard validation tools.

5.1 QNX

QNX [24, 122, 434] is a commercial, Unix-like real-time operating system with POSIX support aimed primarily at the embedded market. Initially released in 1982 for the Intel 8088 and later acquired by BlackBerry it forms the basis of *BlackBerry OS*, *BlackBerry Tablet OS* and *BlackBerry 10* used in mobile devices as well as forming the basis of Cisco's IOS-XR [262] used in carrier-grade routers such as the CRS, the 12000 and the ASR9000 series [258–260]. QNX also dominates the automotive market [234] (particularly telematics, infotainment and navigation systems) and is found in millions of cars from Audi, Toyota, BMW, Porsche, Honda and Ford (among others) as well as being deployed in highly sensitive embedded systems such as industrial automation PLCs, medical devices, building management systems, railway safety equipment, Unmanned Aerial Vehicles (UAVs), anti-tank weapons guidance systems, the Harris Falcon III military radios, Caterpillar mining control systems, General Electric turbine control systems and Westinghouse and AECL nuclear powerplants [222–225].

QNX supports a wide range of CPU architectures and features a pre-emptible microkernel architecture with multi-core support ensuring virtually any component (even core OS components and drivers) can fail without bringing down the kernel. QNX itself has a small footprint but support is available for hundreds of POSIX utilities, common networking technologies (IPv4/IPv6, IPSec, FTP, HTTP, SSH, etc.) and dynamic libraries. As opposed to the monolithic kernel ar-

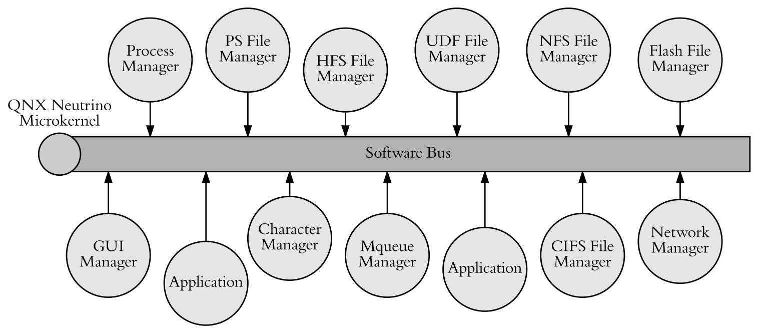


Figure 12: QNX Architecture [233]

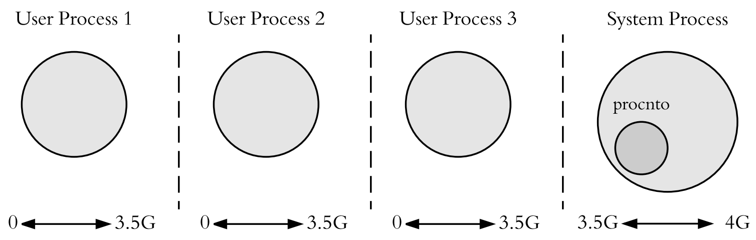


Figure 13: QNX Private Virtual Memory [220]

chitecture of most general-purpose OSes, QNX features a microkernel which provides minimal services (eg. system call and interrupt handling, task scheduling, IPC message-passing, etc.) to the rest of the operating system which runs as a team of cooperating processes as illustrated in Figure 12. As a result, only the microkernel resides in kernelspace with the rest of the operating system and other typical kernel-level functionality (drivers, protocol stacks, etc.) residing in userspace next to regular user applications (though separated by privilege boundaries). In QNX the microkernel is combined with the process manager in a single executable module called `procnto`.

QNX offers a full-protection memory model placing each process within its own private virtual memory by utilizing the MMU as shown in Figure 13. On QNX every process is created with at least one main thread (with its own, OS-supplied stack) and any subsequently created thread can either be given a customly allocated stack by the program or a (default) system-allocated stack for that thread.

In this work, we focus on QNX Neutrino 6.6 which supports all of the exploit mitigations [23] in our baseline: ESP, ASLR and Stack Smashing Protection. The compiler- and linker parts of these mitigations rely on the fact that the QNX Compile Command (QCC) uses GCC as its back-end. QNX, being Unix-like and POSIX-conformant, also offers an operating system random number generator through the `/dev/random` and `/dev/urandom` interfaces [228].

5.1.1 *Security History*

Most of the relatively scarce public research available on QNX security has been the byproduct of research into BlackBerry's QNX-based mobile operating systems such as Tablet OS, BlackBerry OS and BlackBerry 10 [9, 248, 322, 466, 467] most of which has not focussed on QNX itself. Recent work by Plaskett et al. [179, 285] has focussed on QNX itself, particularly security of the Inter-Process Communication (IPC), message passing and Persistent Publish Subscribe (PPS) interfaces as well as kernel security through system call fuzzing. When it comes to specific vulnerabilities the work done by Julio Cesar Fort [68] and Tim Brown [30] stands out in particular and the MITRE CVE database [50] reports, as of writing, 34 vulnerabilities most of which are setuid logic bugs or memory corruption vulnerabilities.

Curiously, a series of documents from the United States *Central Intelligence Agency (CIA)* obtained and released by *WikiLeaks* under the name *'Vault 7'* have shown interest on part of the CIA's *Engineering Development Group (EDG)* (which develops and tests exploits and malware used in covert operations) in targeting QNX [26]. However, branch meeting notes dated October 23, 2014 [253] reveal that while the *Embedded Development Branch (EDB)* has noted QNX as a target of interest on account of its prominent role in vehicle systems, QNX had not been addressed by any *EBD* work at that point.

Since no existing (public) research has evaluated QNX's exploit mitigations, we will present the first reverse engineering analysis and qualitative evaluation of QNX exploit mitigations in the following sections.

5.1.2 *QNX ESP*

Since version 6.3.2, QNX has support for hardware ESP and in Table 6 we list QNX support for hardware ESP among those architectures that both have a hardware ESP feature and that QNX supports. Here we can see that ESP support is absent for MIPS and only present for a subset of PowerPC families.

Insecure QNX ESP Configuration Policy: While QNX supports ESP for several architectures, its implementation is dangerously weakened due to insecure default settings. As a result, the stack (but not the heap) is always executable regardless of hardware ESP support. As the documentation [221] states, the QNX microkernel, and process manager executable (`procnto`) has a memory management startup option relating to stack executability (available as of QNX 6.4.0 or later):

Architecture	**Support**
x86[1]	✓(requires PAE on IA-32e)
ARM	✓
MIPS	×
PPC 400	✓
PPC 600	✓
PPC 900	✓

Table 6: QNX Hardware ESP Support
[1] *Available as of QNX 6.3.2 [226]*

- **-mx**: (Default) Enable the `PROT_EXEC` flag for system-allocated threads (the default). This option allows gcc to generate code on the stack - which it does when taking the address of a nested function (a GCC extension).

- **-m~x**: Turn off `PROT_EXEC` for system-allocated stacks, which increases security but disallows taking the address of nested functions. You can still do this on a case-by-case basis by doing an `mprotect()` call that turns on `PROT_EXEC` for the required stacks.

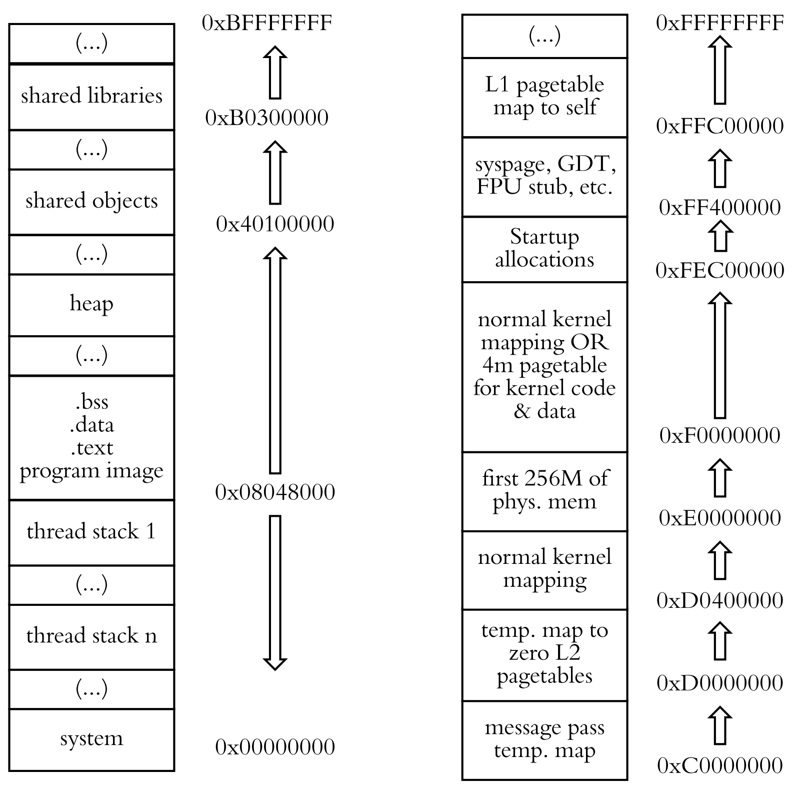
Since the first option is the default any QNX system which starts `procnto` without explicit `-m~x` settings will have an executable stack, regardless of hardware ESP support or individual binary `GNU_STACK` [137] settings. The rationale behind this decision seems to have been a desire for backwards compatibility with binaries which require executable stacks which has caused similar issues on Linux in the past [94]. This backwards compatibility is enforced on a system-wide (rather than on an opt-out, per-binary basis) as confirmed by the fact that the QNX program loader does not parse the `GNU_STACK` header of binaries. The problem with the QNX approach here is that this setting is applied on a system-wide basis and has an insecure default, putting the burden on the end user deploying the system. In order to address this, it is recommended to use an opt-out scheme with secure defaults instead such as the `GNU_STACK` header mechanism.

5.1.3 *QNX ASLR*

QNX supports ASLR as of version 6.5 [218] (not supported for QNX Neutrino RTOS Safe Kernel 1.0) but it's disabled by default. QNX ASLR can be enabled on a system-wide basis by starting the `procnto` microkernel with the `-mr` option [221] and disabled with the `-m~r` option and can be enabled or disabled on a per-process basis by using the `on` utility [219] (with the `-ae` and `-ad` options respectively).

Architecture	User-Space	Kernel-Space
x86	0x00000000 – 0xBFFFFFFF	0xC0000000 – 0xFFFFFFFF
ARM	0x00000000 – 0xBFFFFFFF	0xC0000000 – 0xFFFFFFFF
MIPS	0x00000000 – 0x7FFFFFFF	0x80000000 – 0xFFFFFFFF
PPC	0x40000000 – 0xFFFB0000	0x00000000 – 0x3FFFFFFF
SuperH	0x00000000 – 0x7BFF0000	0x80000000 – 0xCFFFFFFF

Table 7: QNX Address Boundaries



(a) User-Space Layout (b) Kernel-Space Layout

Figure 14: QNX Memory Layout (x86)

Before we discuss the internals of QNX's ASLR implementation and address its quality, we will first consider its memory layout model. Based on reverse-engineering we illustrate QNX user- and kernel-space address boundaries in Table 7 and illustrate user- and kernel-space layouts (when ASLR is disabled) for x86 in Figure 14. On QNX systems where ASLR is not enabled `libc` is loaded by default at the addresses illustrated in Table 8.

The QNX documentation mentions [218] that a child process normally inherits its parent's ASLR setting, but in QNX 6.6 or later you can change it by using the `on` [219] command's `-ae` and `-ad` options

Architecture	Libc Addr.
x86	0xB0300000
ARM	0x01000000
MIPS	0x70300000
PPC	0xFE300000
SuperH	0x70300000

Table 8: QNX Default Libc Load Addresses

to respectively enable and disable it when starting a process from the command line. Alternatively, one can use the `SPAWN_ASLR_INVERT` or `POSIX_SPAWN_ASLR_INVERT` flags with the `spawn` [232] and `posix_spawn` [231] process spawning calls. To determine whether or not a process is using ASLR, one can use the `DCMD_PROC_INFO` [217] command with the `devctl` [229] device control call and test for the `_NTO_PF_ASLR` bit in the flags member of the `procfs_info` structure.

As shown in Table 9, QNX ASLR randomizes the base addresses of userspace and kernelspace **stack**, **heap** and ***mmap'ed*** addresses as well as those of userspace **shared objects** (eg. loaded libraries) and the **executable image** (if the binary is compiled with PIE [23]). It does not, however, have so-called *KASLR* support in order to randomize the kernel image base address. The QNX Momentics Tool Suite [227] development environment (as of version 5.0.1, SDP 6.6) does not have PIC/PIE enabled by default and indeed after an evaluation with a customized version of the *checksec* [118] utility we found that none of the system binaries (eg. those in `/bin`, `/boot`, `/sbin` directories) are PIE binaries in a default installation.

We reverse-engineered QNX's ASLR implementation (as illustrated in Figure 15) and found that it is ultimately implemented in two function residing in the microkernel: `stack_randomize` and `map_find_va` (called as part of `mmap` calls). QNX uses the *Executable and Linking Format (ELF)* binary format and processes are loaded from a filesystem using the `exec*`, `posix_spawn` or `spawn` calls which invoked the program loader implemented in the microkernel. If the ELF binary in question is compiled with PIE-support, the program loader will randomize the program image base address as part of an `mmap` call. When a loaded program was linked against a shared object, or a shared object is requested for loading dynamically, the runtime linker (contained in `libc`) will load it into memory using a series of `mmap` calls. A stack is allocated automatically for the main thread (which involves an allocation of stack space using `mmap`) and has its base address (further) randomized by a call to `stack_randomize`. Whenever a new thread is spawned, a dedicated stack is either allocated (and man-

Memory Object	Randomized
Userspace	
Stack	✓
Heap	✓
Executable Image	✓
Shared Objects	✓
`mmap`	✓
Kernelspace	
Stack	✓
Heap	✓
Kernel Image	×
`mmap`	✓

Table 9: QNX ASLR Memory Object Randomization Support

aged) by the program itself or (by default) allocated and managed by the system in a similar fashion. Userspace and kernelspace heap memory allocation, done using functions such as `malloc`, `realloc` and `free`, ultimately relies on `mmap` as well. In kernelspace, a dedicated stack is allocated for each processor using a call to `_scalloc` and thus relies on `mmap`. As such, all ASLR randomization can be reduced to analysis of `stack_randomize` and `map_find_va`:

map_find_va ASLR: As shown in Listings 2, 3 and 4, the QNX memory manager's `vmm_mmap` handler function invokes `map_create` and passes a dedicated mapping flag (identified only as `MAP_SPARE1` in older QNX documentation) if the ASLR process flag is set. `map_create` then invokes `map_find_va` with these same flags, which randomizes the found virtual address with a randomization value obtained from the lower 32 bits of the result of the `ClockCycles` [214] kernel call. This 32-bit randomization value is then bitwise left-shifted by 12 bits and bitwise and-masked with 24 bits resulting in a value with a mask form of `0x00FFF000`, ie. a randomization value with at most 12 bits of entropy.

stack_randomize ASLR: Userspace processes have a main stack and a dedicated stack for each subsequently spawned thread. The main stack is allocated by the program loader using an `mmap` call and subsequently has its start address randomized by `stack_randomize` called as part of a `ThreadTLS` call. Dedicated thread stacks are spawned by the `thread_specret` routine which allocates them with an `mmap` call (invoked as part of a call to `procmgr_stack_alloc`) and subsequently randomizes their start address with `stack_randomize`. This function, as shown in reverse-engineered form in Listing 5, checks whether a

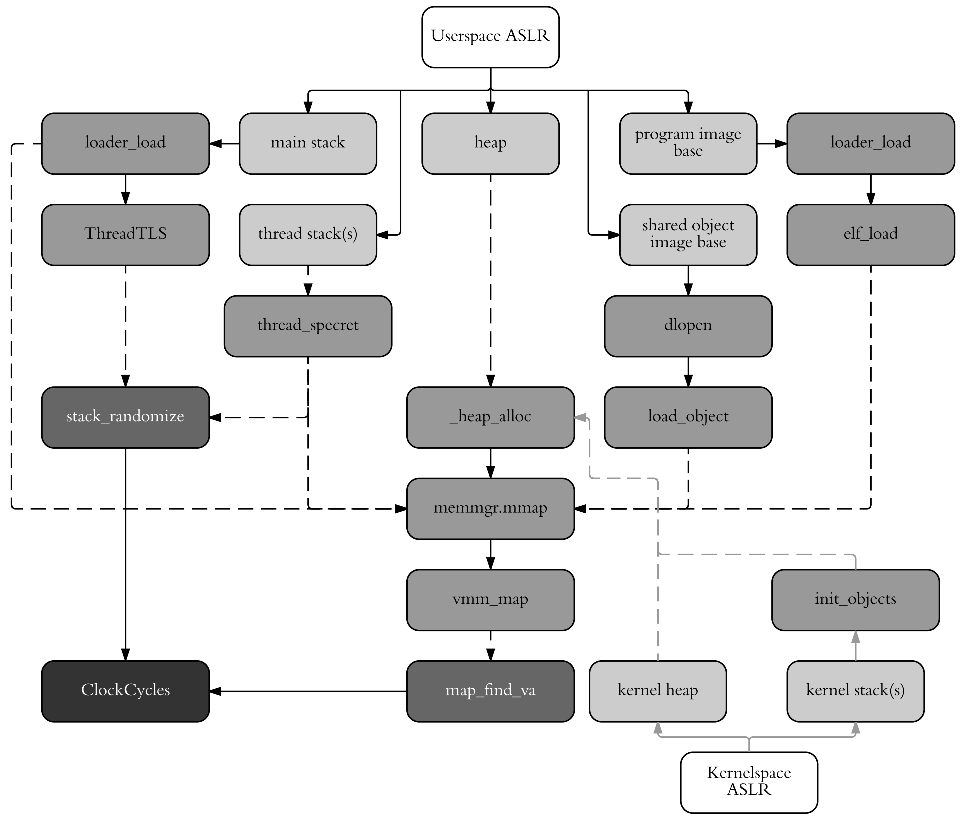


Figure 15: QNX ASLR Memory Object Graph

process has the ASLR flag set and if so it generates a sizemask (between `0x000` and `0x7FF`). A randomization value is drawn from the lower 32 bits of the result of a `ClockCycles` kernel call which are then bitwise left-shifted by 4 bits and have the sizemask applied to them. The resulting value is subtracted from the original stack pointer value and bitwise and-masked with 28 bits. This results of a randomization value with a mask form of `0x000007F0`, ie. 7 bits of entropy for the maximum value of `size_mask = 0x7FF`.

Evaluation: We will proceed to evaluate the QNX ASLR implementation with respect to metrics outlined in Section 3.2.2.

Weak ASLR Randomization (CVE-2017-3892): As observed above, the randomization underlying `mmap` introduces at most 12 bits of entropy and the additional randomization applied to userspace stacks introduces at most 7 bits of entropy, combining into at most 19 bits of entropy with a mask of form `0x00FFF7F0`. Addresses with such low amounts of entropy can be easily bruteforced (in mere seconds or minutes locally and minutes or hours remotely) and while ASLR on 32-bit systems is generally considered inherently limited [356] one should remember that these are entropy *upper bounds*, ie. they express the maximum possible introduced entropy. Given that these

Architecture	Implementation
x86	*RDTSC* [105]
ARM	Emulation
MIPS	*Count Register* [103]
PPC	*Time Base Facility* [162]
SuperH	*Timer Unit (TMU)* [198]

Table 10: QNX *ClockCycles* Implementations

upper bounds already compare rather unfavorably against the measurements of *actual* ASLR entropy in Linux 4.5.0, PaX 3.14.21 and ASLR-NG 4.5.0 as per [356], this does not bode well for once we take the actual entropic quality into account as well. Both QNX ASLR randomization routines draw upon `ClockCycles` as the sole source of entropy. The QNX `ClockCycles` [214] kernel call returns the current value of a free-running 64-bit cycle counter using a different implementation per processor and emulating such functionality when it's not present in hardware as outlined in Table 10. Even though QNX's usage of clock cycles seems to provide 32 bits of 'randomness', it is an ill-advised source of entropy due to its inherent regularity, non-secrecy, and predictability.

In order to demonstrate this, we evaluated the entropic quality of QNX ASLR randomized addresses of several userspace memory objects. We did this with a script starting 3000 ASLR-enabled PIE processes logging the relevant address values per boot session and running 10 boot sessions, collecting 30000 samples per memory object in total. We used the *NIST SP800-90B* [158] *Entropy Source Testing (EST) tool* [155] in order to evaluate the entropic quality of the address samples by means of a *min entropy* [255] estimate, illustrated in Table 11. Min entropy is a conservative way of measuring the (un)predictability of a random variable X and expresses the number of (nearly) uniform bits contained in X, with 256 bits of uniformly random data corresponding to 256 bits of min entropy. Being the smallest entropy measure in the Rényi [256] family, H_∞, min entropy is the strongest (un)predictability measure of a discrete random variable.

From Table 11 we can see that, on average, a QNX randomized userspace memory object has a min entropy of `1.11785975`. This means that it has a little more than 1 bit of min entropy per 8 bits of data. If we extrapolate this to the full 32 bits of a given address this means that the stack, heap, executable image and shared object base addresses have respectively min entropy values of `6.39944`, `4.03656`, `3.827172` and `3.622584`, with an average of `4.471439` bits of min entropy. This compares very unfavorably with the entropy measure-

Memory Object	Min Entropy (8 bits per symbol)
Stack	1.59986
Heap	1.00914
Executable Image	0.956793
Shared Objects	0.905646

Table 11: QNX ASLR Userspace Memory Object Min Entropy

ments for various Linux-oriented ASLR mechanisms in [356].

ASLR Brute-Force Susceptibility (CVE-2017-3892): On QNX, as is the case with many operating systems, child processes inherit the memory layout of parent processes. As explained in Section 3.2.2, this increases susceptibility to both brute-force attacks and malicious child processes attacking siblings in *Android Zygote*-style sandboxing models. Given QNX's memory layout inheritance and the fact that its ASLR implementation provides limited entropy and has no active relocation (ie. memory object locations are randomized when they are mapped and never re-randomized), QNX ASLR is susceptible to brute-force attacks under the circumstances outlined in Section 3.2.2.

ASLR Information Leaks: QNX's randomization routines don't use a PRNG but directly draw output from the entropy source. As such the clock cycle counter value becomes more or less equivalent to a PRNG internal state and should thus be both *secret* and *unpredictable*, but unfortunately it is neither. Reading the clock cycle counter is not a privileged operation and thus any local attacker can simply obtain the current value. In addition, since clock cycle counter values are usually not considered secret, it is not uncommon for them to be leaked to remote attackers either (eg. directly as part of public system information or implicitly when utilized for generation of some public randomized value). Alternatively, a remote attacker who is able to learn or approximate a target system's time of boot (eg. via services or protocols which broadcast uptime or through techniques such as those suggested in [151]) might be able to infer the current clock cycle counter value from this. Finally, since the clock cycle counter is effectively a system-wide variable this means that it is identical across applications (but not across CPUs for SMP systems) and as such an information leak in one application disclosing this value might be utilized against a completely different application.

Apart from this potential general information leak concerning QNX ASLR entropy, we also discovered several system-wide information leaks disclosing randomized addresses that could be used to circumvent QNX ASLR:

- **procfs Infoleak (CVE-2017-3892)**: The proc filesystem (`procfs` [267]) is a pseudo-filesystem on UNIX-like operating systems that contains information about running processes and other system aspects via a hierarchical file-like structure. This exposure of process information often includes ASLR-sensitive information (eg. memory layout maps, individual pointers, etc.) and as such has a history as a source for local ASLR infoleaks [206, 238, 452] with both *GrSecurity* [475] and mainline Linux [57, 244] seeking to address procfs as an infoleak source. On QNX `procfs` [216] is implemented by the process manager component of `procnto` and provides the following elements for each running process:

  - **cmdline**: Process command-line arguments.
  - **exefile**: Executable path.
  - **as**: The virtual address space of the target process mapped as a pseudo-file.

  These procfs entries can be interacted with like files and subsequently manipulated using the `devctl` [229] API to operate on a file-descriptor resulting from opening a `procfs` PID entry. Since process entries in QNX's `procfs` are world-readable by default, this means a wide range of `devctl`-based information retrieval about any process is available to users regardless of privilege boundaries. For example, the QNX `pidin` [230] utility, which makes use of `procfs` to provide a wide range of process inspection and debugging options, easily allows any user to obtain stackframe backtraces, full memory mappings and program states for any process. This effectively constitutes a system-wide local information leak allowing attackers to circumvent ASLR. It should be noted this issue is not due to the availability of any particular utility (such as `pidin`) but rather results from a fundamental lack of privilege enforcement on `procfs`.

- **LD_DEBUG Infoleak (CVE-2017-9369)**: The `LD_DEBUG` [167] environment variable is used on some UNIX-like systems to instruct the dynamic linker to output debug information during execution. On QNX there are no cross-privilege restrictions on this environment variable, leading to a (local) information leak that can be used to circumvent ASLR. Since there are no privilege checks (akin to the *'secure-execution mode'* [243] offered by some Linux distributions) on this environment variable, a local attacker can execute setuid binaries with higher privileges using dynamic linker debugging settings (eg. `LD_DEBUG=all`) in order to output sensitive information (eg. memory layout, pointers, etc.) which can be used to circumvent ASLR. This issue is analogous to `CVE-2004-1453` [33] affecting certain versions of GNU glibc.

ASLR Correlation Attacks (CVE-2017-3892): As discussed in Section 3.2.2, ASLR randomization of memory object base addresses can prove to be insufficient if different memory objects are correlated. During our evaluation we found a *partial* correlation attack on QNX's ASLR implementation, affecting both PIE and non-PIE binaries. The offset from the program image base to the base address of the first loaded shared library (`libc`) is of the mask form `0x00FFF000` with at most 12 bits being randomized. We evaluated the entropic quality of this offset value in order to determine the feasibility of correlation attacks by collecting 300 offset samples per boot session and running 10 boot sessions, making for 3000 samples total. Using the *NIST Entropy Source Testing (EST) tool* [155] we determined the min entropy of these offset values to be `0.918311`, making for less than 1 bit of min entropy per 8 bits of data, which corresponds to `1.3774665` bits of min entropy for the 12 affected variable bits in the offset. Given that this is well below an exhaustive search, this makes a variation of the *offset2lib* [355] attack (where we perform a minor bruteforce against the offset instead of assuming a constant) feasible.

5.1.4 *QNX Stack Canaries*

QNX's QCC offers stack canary support in the form of the GCC Stack Smashing Protector (SSP) [190] and supports all the usual SSP flags (*strong*, *all*, etc.). Since the compiler-side of the QNX SSP implementation is identical to the regular GCC implementation, the *master canary* is stored accordingly and canary violation invokes the `__stack_chk_fail` handler.

For *user-space applications*, this handler is implemented in QNX's *libc*. On the OS-side, reverse-engineering of *libc* shows us that violation handler (shown in cleaned-up form in Listing 6) is a wrapper for a custom function called *_ssp_fail* which writes an alert message to the `/dev/tty` device and raises a `SIGABRT` signal.

QNX generates the master canary values once upon program startup (during loading of *libc*) and is not renewed at any time. Instead of the regular `libssp` function `__guard_setup`, QNX uses a custom function called `__init_cookies` (shown in Listing 7) invoked by the *_init_libc* routine in order to (among other things) generate the master canary value.

When it comes to *kernel-space stack canary protection*, the QNX microkernel (in the form of the *procnto* process) also features an SSP implementation covering a subset of its functions. Since the kernel neither loads nor is linked against *libc* (and canary violations need to be handled differently), SSP functionality is implemented in a custom fashion here. We reverse-engineered the microkernel and found that it has a custom `__stack_chk_fail` handler (illustrated in Listing 8) but no master canary initialization routine (as discussed below).

Insecure QNX User-Space Canary Generation: As shown in Listing 7 QNX SSP canaries have a terminator-style NULL-byte in the middle (as per the applied bitmask) and are generated using a custom randomization routine (slightly resembling the *"poor man's randomization patch"* included in some Linux distrubtions for performance purposes [76, 77]) rather than drawing it from a CSPRNG source. The custom randomization routine draws upon three sources:

- **_init_cookies**: This is the function's own address and as such, the only randomization it introduces is derived from ASLR's effect on shared library base addresses which means that if ASLR is disabled (or circumvented with an infoleak) this is a per-glibc static value.
- **stackval**: This is an offset to the current stack pointer and as such, the only randomization it introduces is derived from ASLR's effect on the stack base which means that if ASLR is disabled (or circumvented with an infoleak) this is simply a static value known to the attacker.
- **ClockCycles**: The lower 32 bits of the result of a `ClockCycles()` call are used to construct the master canary value.

Since it includes a terminator-style NULL-byte, the QNX master stack canary value has at most 24 bits of entropy. When ASLR is disabled on a target system (the default for QNX) the `_init_cookies` and `stackval` address values contribute no entropy. If ASLR is enabled, however, they contribute at most 12 and 19 bits of entropy respectively as discussed in Section 5.1.3. All entropy in QNX stack canaries, however, is ultimately based on invocations of the `ClockCycles` kernel call. If ASLR is enabled, the `stackval` address gets randomized when the main thread is spawned during program startup and the `_init_cookies` address gets randomized when *libc* gets loaded by the runtime linker. The `ts0` value is generated when `_init_cookies` is called by *_init_libc* which is invoked upon application startup (but after *libc* is loaded).

In order to evaluate the entropic quality of QNX's stack canary generation and investigate the extent to which ASLR contributes to it, we

Settings	**Min Entropy (8 bits per symbol)**
No ASLR	1.94739
ASLR, no PIE	1.94756
ASLR + PIE	1.94741

Table 12: QNX Stack Canary Min Entropy

collected canary values for three different process configurations: No ASLR, ASLR but no PIE and ALSR with PIE. We used a script starting 785 instances of each configuration per boot session and repeated this for 10 boot sessions, collecting 7850 samples per configuration in total. We then used the *NIST Entropy Source Testing (EST) tool* [155] in order to obtain min entropy estimates for the sample sets as illustrated in Table 12. Based on these observations we can conclude that a) QNX stack canary entropy is far less than the hypothetical upper bound of 24 bits, being on average `7.78981332` bits for a 32-bit canary value and b) ASLR plays no significant contributing role to the overall QNX stack canary entropy.

An attacker with an information or sidechannel leak (as discussed in Section 5.1.3) disclosing the current clock cycle counter value, could be utilized by an attacker to mount an attack potentially faster than brute-force. Given the current clock cycle counter value and an estimate on memory object initialization times, an attacker can deduce the clock cycle counter value at randomization time for a given memory object and reconstruct it as:

$$ccv_t = ccv_{curr} - ((time_{curr} - time_t) * cycles_{sec})$$

where $ccv_t, ccv_{curr}, time_t$ and $time_{curr}$ are the target and current clock cycle counter and timestamp values and $cycles_{sec}$ is the number of cycle increments per second. An attacker could use these reconstructed clock cycle counter values to reconstruct the *master canary value* at creation time faster after a certain number of attempts, thus reducing attack complexity from a full 24-bit brute-force to a brute-force determined by our *'window of uncertainty'*. The practicality of this approach depends on two factors: a) *our timing precision* and b) *the target system's clock speed*. The former is relevant because we will have to brute-force within any *'window of uncertainty'* that we have regarding target randomization time and this is easier for some memory objects (eg. the program image and main stack base addresses) than others (eg. thread stack and heap addresses). The latter is relevant because QNX uses the 32 least significant bits of the clock cycle counter for randomization. As such we need to take into account that this value might *'wrap around'* several times. The speed at which this value *'wraps around'* is determined by the clock speed, eg. a 700 MHz CPU will cause a wraparound of the LSBs in roughly $\frac{2^{32}=4294967295}{700000000} \approx 6.14$ seconds. At high clock speeds, the lower 32 bits of the counter will wrap around too fast for reasonable attacker timing granularity requirements, forcing an attacker to mount a brute-force attack on the full search-space instead. Luckily for attackers, embedded systems tend to feature CPUs running at lower clockspeeds.

Canary Brute-Force Susceptibility: Given the low entropic quality of QNX stack canary values, its memory layout inheritance and the fact that QNX's SSP implementation only generates master canary values once during program startup and never renews them, an attacker could, under circumstances outlined in Section 3.2.3, mount a brute-force attack on the stack canary [354]. When it comes to *byte-for-byte brute-force attacks* [273, 354], this comes with the caveat that it is unsuitable when exploiting certain string operations (eg. `strcpy`) because of the NULL-terminator included in the QNX stack canary.

Absent QNX Kernel-Space Canary Generation: Since the QNX microkernel uses but never actually initializes the master canary value it is always all-zero (ie. `0x00000000`) and thus completely static and known to the attacker, making it trivial to bypass and defeating the purpose of stack canary protection for QNX kernel-space code.

5.1.5 *QNX OS CSPRNG*

QNX provides secure random number generation by means of the *random* service [228] which runs as a userspace process started by the `/etc/rc.d/startup.sh` script and addressed by the kernel resource manager. It exposes its output through the `/dev/random` and `/dev/urandom` interfaces, both of which are non-blocking in practice. The CSPRNG underlying the QNX *random* service is based on the *Yarrow* CSPRNG [373] (which is also used by iOS, Mac OS X, AIX and some BSD descendants) by John Kelsey, Bruce Schneier and Niels Ferguson rather than its recommended successor *Fortuna* [414].

The *QNX Yarrow* implementation (as illustrated in Figure 16), however, is not based on the reference *Yarrow-160* [373] design but instead on an older *Yarrow 0.8.71* implementation by Counterpane [45] which has not undergone the security scrutiny Yarrow has seen over the years and differs in the following key aspects:

- **Single Entropy Pool**: While *Yarrow-160* has separate fast and slow entropy pools, *Yarrow 0.8.71* only has a single entropy pool. The two pools were introduced so that the fast pool could provide frequent reseeds of the *Yarrow* key to limit the impact of state compromises while the slow pool provides rare, but very conservative, reseeds of the key to limit the impact of entropy

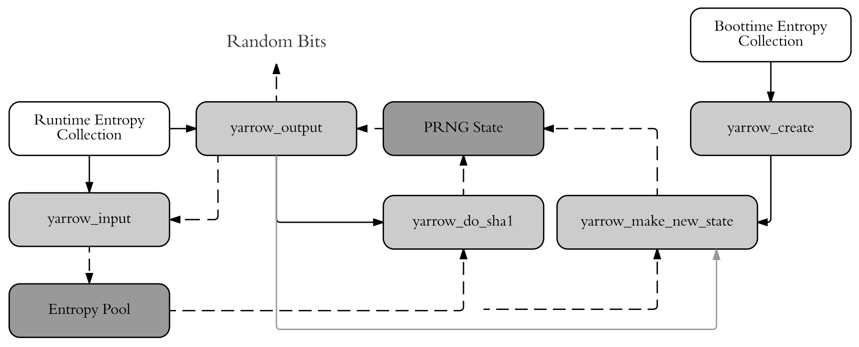


Figure 16: Simplified QNX Yarrow 6.6 Design

estimates which are too optimistic. *Yarrow 0.8.71*'s single pool does not allow for such security mechanisms.

- **No Blockcipher Applied To Output**: As opposed to *Yarrow-160*, *Yarrow 0.8.71* does not apply a block cipher (eg. in CTR mode) to the *Yarrow* internal state before producing PRNG output and instead simply outputs the internal state directly which results in a significantly weaker design than that of *Yarrow-160*.

In addition, the *QNX Yarrow* implementation diverges from *Yarrow 0.8.71* as well in the following aspects:

- **Mixes PRNG Output Into Entropy Pool**: As part of its various entropy collection routines, *QNX Yarrow* mixes PRNG output back into the entropy pool. For example in the high performance counter entropy collection routine (as per the snippet in Listing 9) we can see PRNG output is drawn from *QNX Yarrow*, used as part of a delay routine and subsequently mixed (via a xor operation with the result of a `ClockCycles` call) back into the entropy pool. This construction deviates from all *Yarrow* designs and while not a crucial flaw, it is ill-advised in the absence of further security analysis or justification.

- **Absent Reseed Control (QNX < 6.6.0)**: In all QNX versions prior to 6.6.0 reseed control is completely absent. While the required functionality was implemented, the responsible functions are *never actually invoked*, which means that while entropy is being accumulated during runtime it is never actually used to reseed the state and thus only boottime entropy is actually ever used to seed the *QNX Yarrow* state in versions prior to 6.6.0.

- **Custom Reseed Control (QNX 6.6.0)**: In QNX 6.6.0 there is a custom reseeding mechanism integrated into the `yarrow_do_sha1` and `yarrow_make_new_state` functions (as illustrated in Listings 10 and 11) which are called upon PRNG initialization and whenever output is drawn from the PRNG (which means it is also constantly called during entropy accumulation due to the above mentioned output mixing mechanism). In both cases, a permutation named `IncGaloisCounter5X32` is applied to the entropy pool before the pool contents are mixed into a SHA1 state which eventually becomes the Yarrow internal state. Contrary to *Yarrow* design specifications, no entropy quality estimation is done before reseeding.

While all the above discussed divergences are at the very least ill-advised, the reseeding control issues constitute a clear security issue. In the case of absent reseeding control, it eliminates *Yarrow*'s intended defense against state compromise as well as greatly increasing system susceptibility to the so-called *"boottime entropy hole"* [407] that affects embedded systems. In the case of the *QNX Yarrow 6.6.0* custom reseeding control no entropy quality estimation is done before reseeding the state from the entropy pool thus potentially allowing for low-quality entropy to determine the entire state.

In order to evaluate the *QNX Yarrow* PRNG output quality we used two test suites: *DieHarder* [29] and the *NIST SP800-22* [157] *Statistical Test Suite (STS)* [156]. *DieHarder* is a random number generator testing suite, composed of a series of statistical tests, *"designed to permit one to push a weak generator to unambiguous failure"* [29]. The *NIST Statistical Test Suite (STS)* consists of 15 tests developed to evaluate the 'randomness' of binary sequences produced by hardware- or software-based cryptographic (pseudo-) random number generators by assessing the presence or absence of a particular statistical pattern. The goal is to *"minimize the probability of accepting a sequence being produced by a generator as good when the generator was actually bad"* [157]. While there are an infinite number of possible statistical tests and as such no specific test suite can be deemed truly complete, they can help uncover particularly weak random number generators.

QNX Yarrow passed both the *DieHarder* and *NIST STS* tests but this only tells us something about the quality of PRNG output, leaving the possibility open that raw noise / source entropy is (heavily) biased which can result in predictable PRNG outputs as well as attackers being able to replicate PRNG internal states after a reasonable number of guesses. As such we reverse-engineered and evaluated the QNX *random* service's boot- and runtime entropy sources.

Boottime Entropy Analysis: When *random* is initialized it gathers initial boottime entropy from the following sources (as illustrated in Figure 17) which are fed to the SHA1 hash function to produce a digest used to initialize the PRNG initial state:

- **ClockTime** [215]: The current system clock time.

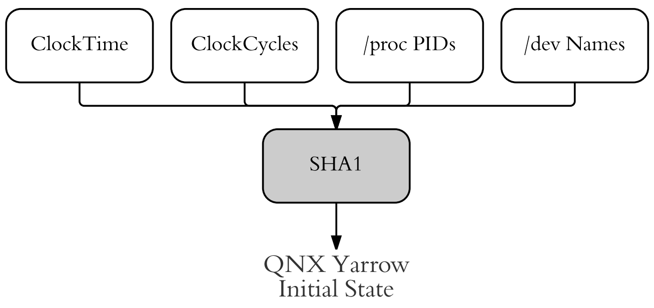


Figure 17: QNX Yarrow Boottime Entropy Collection

- **ClockCycles** [214]: The current value of a free-running 64-bit clock cycle counter.
- **PIDs**: The currently active process IDs by walking the `/proc` directory.
- **Device Names**: The currently available device names by walking the `/dev` directory.

In order to evaluate *random*'s boottime entropy quality we used the *NIST SP800-90B* [158] *Entropy Source Testing (EST) tool* [155] to evaluate boottime entropy by means of a *min entropy* [255] estimate. We collected *random*'s boottime entropy from 50 different reboot sessions on the same device (by instrumenting `yarrow_init_poll` and logging the collected raw noise) and using *NIST EST* determined that the average min-entropy was `0.02765687`, which is far less than 1 bit of min-entropy per 8 bits of raw noise. In addition to the boottime entropy of individual boot sessions being of low quality, the static or minimally variable nature of many of the boottime noise sources (identical processes and devices available upon reboot, real-time nature of QNX limiting jitter between kernel calls thus reducing `ClockCycles` entropy, etc.) results in predictable and consistent patterns across reboots as shown in Figure 18 which visualizes (with darker sports representing lower entropy) a concatenation of boot-time entropy samples from 50 different reboots.

Another boottime entropy issue with QNX's *random* service is the fact that the service is started as a process by `startup.sh`. As a result, the CSPRNG is only available quite late in the boot process and many services which need it (eg. sshd) start almost immediately after. Since *random* only offers non-blocking interfaces, this means that one can draw as much output from the CSPRNG as one wants immediately upon availability of the device interface. Hence, many applications which start at boot and require secure random data have their 'randomness' determined almost completely by the (very low quality)

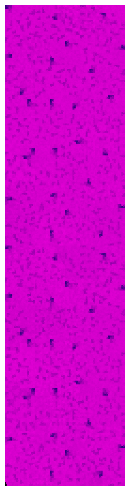


Figure 18: QNX Yarrow Restart Boottime Entropy Visualization [43]

boottime raw noise since there is little time for the QNX *random* service to gather runtime entropy before being queried thus amplifying the impact of the *"boot-time entropy hole"* [407].

Runtime Entropy Analysis: The QNX *random* service leaves the choice and combination of runtime entropy sources (as illustrated in Figure 19) up to the person configuring the system with the following options:

- **Interrupt Request Timing**: Up to 32 different interrupt numbers may be specified to be used as an entropy source. The entropy here is derived from interval timing measurements (measured by the `ClockTime` kernel call) between requests to a specific interrupt.
- **System Information Polling**: This source collects entropy from system information via the *procfs* [216] virtual filesystem in `/proc`. This information is composed of process and thread information (process and thread IDs, stack and program image base addresses, register values, flag values, task priority, etc.) for every currently active process.
- **High-Performance Clock Timing**: This source draws entropy from the PRNG (using the `yarrow_output` function), initiates a delay (in milliseconds) based on the PRNG output, invokes `ClockCycles` and xors the result against the earlier obtained PRNG output and feeds this into the entropy pool.
- **Library Hardware Entropy Source (Undocumented)**: This undocumented entropy source (invoked using command-line parameter *-l*) allows a user to specify a dynamic library to supply entropy collection callback functions named `entropy_source_init`

and `entropy_source_start`. In order to be used the library has to export a symbol named `cookie` with the NULL-terminated value `RNG` (`0x524E4700`). Based on debugging information it seems this is to allow for drawing from a hardware random number generator as an entropy source.

- **User-Supplied Input (Undocumented)**: In QNX 6.6 the *random* service has a write-handler made available to users via the kernel resource manager (in the form of handling write operations to the `/dev/(u)random` interfaces) which takes arbitrary user inputs of up to 1024 bytes per write operation and feeds it directly into the entropy pool by passing it to the `yarrow_input` operation. Write operations of this kind are restricted to the root user only.

After initialization, *random* starts a thread for each entroy source which will gather entropy and store it in the entropy pool. Contrary to our analysis of QNX *random*'s boottime entropy, we did not perform a runtime entropy quality evaluation because during our contact with the vendor (as part of a responsible disclosure process) they had already indicated the current design would be overhauled in upcoming patches and future releases as a result of our findings with regards to the *QNX Yarrow* design and boottime entropy issues. In addition, in all QNX versions except 6.6 runtime entropy is accumulated but not used due to the previously mentioned absent reseeding control. We did have the following observations however:

- **Entropy Source Configuration**: Configuring runtime entropy sources is entirely left to system integrators. Since the entropic quality of certain sources (eg. interrupt request timings or system information polling) varies depending on the particular system, it is non-trivial to pick suitable sources.

- **System Information Entropy Source**: System information polling gathers raw noise from currently running processes (in the form of process and thread debug info). A significant number of the fields in the process and thread debug info structures, however, are largely static values (eg. uid, flags, priority, stack and program base in the absence of ASLR, etc.) with most randomness derived from time-based fields (starttime, runtime) or program state (ip, sp).

- **Interrupt Request Timing Entropy Source**: Interrupt request timing gathers raw noise from interrupt invocation timings. As such this means that if integrators choose to specify interrupts that are rarely or never invoked, barely any runtime entropy is gathered using this source. Interrupt invocation frequency can be very system specific and picking the right interrupts is not

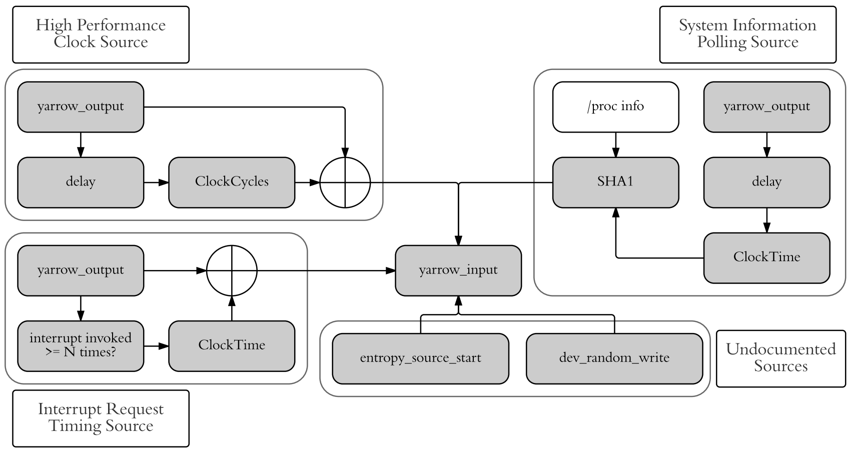


Figure 19: QNX Yarrow Runtime Entropy Collection

trivial. The QNX documentation explicitly recommends to *"minimize the impact of [interrupt request timing overhead] by specifying only one or two interrupts from low interrupt rate devices such as disk drivers and input/serial devices"* [228], an advice which would result in less entropy being accumulated from this source. Furthermore, it seems that if for whatever reasons the *random* service cannot attach to an interrupt, the interrupt entropy gathering thread fails silently and no entropy is gathered for that interrupt at all.

5.2 REDACTEDOS

Due to a *non-disclosure agreement* with the vendor we cannot name the operating system in question and as such will refer to it as `RedactedOS` throughout this work. `RedactedOS` is a commercial, Unix-like, POSIX-conformant, real-time operating system aimed at the embedded market which supports a wide range of CPU architectures, has a monolithic (but modular) kernel architecture with multi-core support as well as support for common networking technologies and applications. `RedactedOS` is used in highly sensitive systems such as civilian jet airliners, multirole combat aircrafts, military radio systems and space station equipment. In this work we focus on the latest release of `RedactedOS` as of writing. Our research found `RedactedOS` to not support any of the exploit mitigations in our minimum baseline as per Section 3.1 and as such we will focus solely on the operating system CSPRNG. There is no (publicly available) security history of `RedactedOS` .

5.2.1 *RedactedOS OS CSPRNG*

`RedactedOS` provides a Unix-style OS pseudo-random number generator used for cryptographic purposes via the `/dev/(u)random` interfaces, both of which are non-blocking. The PRNG has a read handler which returns `N` random bytes generated by the internal `random` function, and a write handler which reseeds the PRNG using the first 4 bytes drawn from a user-supplied buffer. We found that the `RedactedOS` PRNG is insecure in its general design as well as suffering from several specific implementation issues and as such is unsuitable for its current cryptographic use as well as any future mitigation supporting purposes. The potential impact here is significant as it allows for the potential compromise of the entire `RedactedOS` cryptographic ecosystem (eg. many 3rd party software packages such as OpenSSL draws rely on `/dev/urandom` for secure randomness on Unix-like systems) by either local or remote attackers. Addressing these issues requires a complete redesign of the `RedactedOS` /dev/urandom PRNG. More precisely, we found the following issues:

Insecure PRNG Algorithm: After inspection, we found that the algorithm underlying the `RedactedOS` PRNG is the GNU libc BSD `random` PRNG [175] with modified constants for the initial state, linear congruential and lookahead parts. This PRNG, as explicitly stated in the man pages, is not a cryptographically secure PRNG but a so-called additive congruential pseudo-random number generator [268]. Using an insecure PRNG means that an attacker who learns a certain number of PRNG outputs might be able to reconstruct the corresponding PRNG internal state [295] (and hence predict all future PRNG outputs) or even 'roll back' the internal state to recover the original seed [192] (and hence reconstruct past PRNG outputs as well) as shown by attacks on insecure PRNGs commonly (mis)used for security purposes [112, 347].

Local Reseed Attack: To make matters worse, the `RedactedOS` PRNG is vulnerable to a local reseeding attack. The write handler for the `/dev/(u)random` interface allows for reseeding of the PRNG (there is no reseed control) where a single reseed operation completely determines the internal state for the system-wide PRNG. Given that the `/dev/(u)random` interface is *world-writable* by default this allows user to force a PRNG reseed across privilege boundaries and thus completely control PRNG output drawn upon by privileged users, potentially allowing for compromise of cryptographic or otherwise security sensitive data.

Known Seed Attack: Finally, the `RedactedOS` PRNG is vulnerable to a known seed attack exploitable by both local and remote attackers.

After inspection, we found that not only does the `RedactedOS` PRNG use an insecure PRNG, it uses a single static seed and doesn't implement any form of reseed control meaning *there is no actual entropy in the system at any time and PRNG output sequences are identical upon every new boot*. Since PRNGs are deterministic this means that knowledge of a PRNG seed is equivalent to knowledge of a secret key and allows an attacker to reproduce all PRNG outputs generated from the start (boottime) without having to mount a cryptanalytic attack on the algorithm to recover the internal state. Needless to say, this completely compromises the `RedactedOS` PRNG. Consider, for example, the following scenario (as illustrated in Figure 20):

1. Consider a public-private keypair (eg. RSA or DSA) generated using the `RedactedOS` PRNG: (K_{pub}, K_{priv})
2. A remote attacker has access to public key K_{pub} and initial `RedactedOS` PRNG seed S
3. The attacker clones the `RedactedOS` PRNG and initializes it with seed value S and sets an index value I to 0
4. The attacker advances the PRNG state offset to position I (measured in bytes read from the PRNG interface)
5. The attacker generates a public-private keypair (L_{pub}, L_{priv}) by drawing from the cloned PRNG
6. The attacker compares L_{pub} to K_{pub}: if they mismatch increment I by 1 and go to step 4, if they match we've found our target private key $K_{priv} = L_{priv}$

The only unknown element in the above attack is the PRNG state offset at the time of target key generation, which varies depending on how many bytes have previously been read from the PRNG interface. Given that this number can be reasonably assumed to lie below 2^{32} (ie. ~ 4.29 GB), the exhaustive search in the above attack is very practical.

5.3 ZEPHYR

Initially released in February 2016, Zephyr [164, 186] is an open source, real-time operating system developed by the Zephyr Project as a collaboration project hosted by the Linux Foundation [69]. The Zephyr kernel is derived from Wind River's commercial VxWorks microkernel profile (in turn evolved from the Virtuoso DSP RTOS [104]) and is identical to the Wind River Rocket RTOS kernel [176]. Zephyr is optimized for and aimed primarily at resource constrained devices

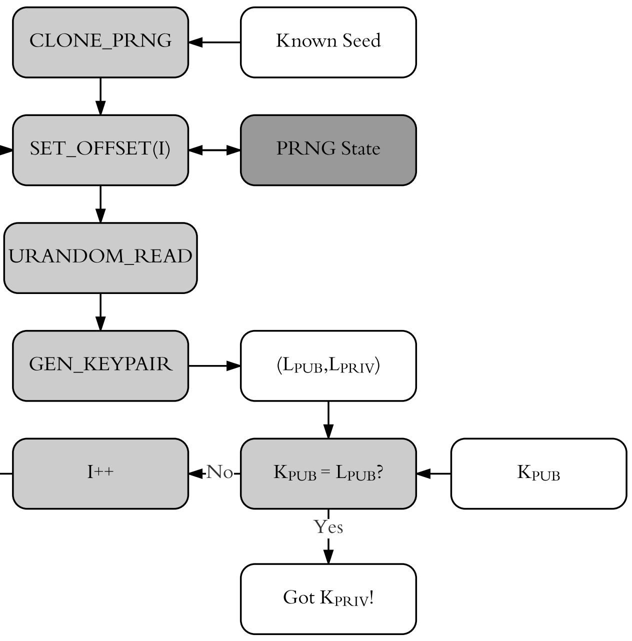


Figure 20: RedactedOS PRNG Known Seed Attack

(eg. wireless sensor nodes, connected light bulbs, wearables, small IoT gateways, etc.) within the Internet of Things and has a minimal code and memory footprint. Zephyr supports multiple architectures (eg. x86, ARM, ARC and NIOS II) and is designed to be very modular so that almost every feature can be enabled or disabled to fit particular device needs and constraints. In addition, Zephyr provides support for various IoT-oriented connectivity protocols such as Bluetooth, Bluetooth Low Energy, Wi-Fi, 802.15.4, 6Lowpan, CoAP, IPv4, IPv6, and NFC. In this work, we focus on Zephyr 1.7 but do include an evaluation of the boards supported by the upcoming Zephyr 1.8 release in Table 27.

As illustrated in Figure 21, Zephyr has a *'two kernel'* design. The essential RTOS services reside within the nanokernel, which is responsible for executing top priority activity and runs a series of minimal threads named *fibers*, while the full microkernel allows for more extensive functionality such as multitasking abilities, memory management, etc. Zephyr is implemented as a *'library-based RTOS'* where all code runs in a single address-space (without virtual memory support but with upcoming memory protection support [163]) and application code is combined with kernel code to create a single monolithic image without runtime loading. System calls are implemented as simple function calls. While the Zephyr Project is explicitly committed to security [186] and offers stack canary support, there is currently no support for ESP or ASLR nor does it offer an OS CSPRNG. There

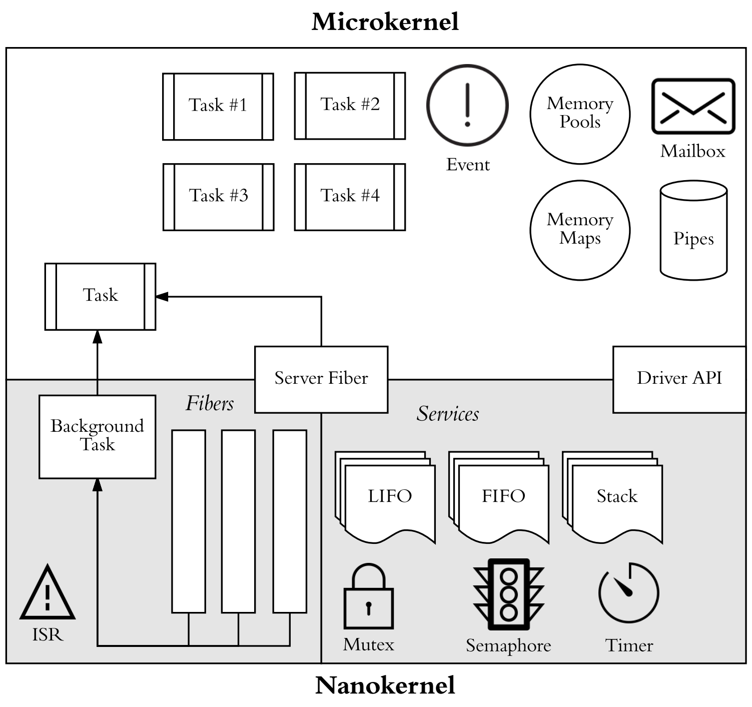


Figure 21: Zephyr Architecture [186]

is no security history for Zephyr which is, given the project's youth, quite understandable.

5.3.1 *Zephyr Stack Canaries*

Zephyr supports GCC as the default compiler and as such follows the GCC SSP [190] canary model. Analysis of the Zephyr source-code shows that the `__stack_chk_fail` handler is implemented as an alias for the `_StackCheckHandler` routine which invokes a fatal software error handler in the nanokernel. This handler displays a stack check failure message and debug log if kernel printing output is enabled and invokes the system fatal error handler `_SysFatalErrorHandler` which by default attempts to abort the current thread and allows the system to continue executing but can be modified by system designers to take other actions (eg. logging, rebooting, etc.).

There is one `__stack_chk_guard` *master canary value* for the entire operating systems, shared among all tasks, which is stored in the `.noinit` data section right above `_interrupt_stack`. Zephyr's SSP implementation protects both the user application as well as *Interrupt Service Routines (ISRs)* written in C and all kernel functionality except for the most rudimentary and early-boot ones (ie. those prior to `_Cstart`).

Random Number Generator Issues: Generation of the master canary is done as part of the *_Cstart* routine (shown in Listing 12) which is invoked as part of the nanokernel initialization when the system is ready to run C code. The *master canary* is generated by the Zephyr random number generation API `sys_rand32_get`, the implementation of which depends on the settings of the *random* driver `Kconfig` file. Zephyr allows for the following RNG configuration options which determine the implementation of `sys_rand32_get`:

- `RANDOM_GENERATOR`: This option signals random drivers should be included in the system configuration and has the the following *'sub-options'*:
  - `TEST_RANDOM_GENERATOR`: This option signifies that the kernel's random number APIs are permitted to return values that are not truly random. While it is explicitly stated in the documentation and source-code that this should only be enabled for testing purposes, it is the only alternative to implementing a custom OS CSPRNG in the absence of hardware / *true* RNG support. This option has two *'sub-options'*:
    - `X86_TSC_RANDOM_GENERATOR`: This option enables number generation based on the x86 timestamp counter (TSC). The random API function becomes a call to `_do_read_cpu_timestamp32` which wraps the x86 `rdtsc` instruction [105] to return the lower 32 bits (stored in the `EAX` register) of the TSC. Given the real-time and minimal nature of Zephyr and the fact that the *master canary* value is generated during system startup (directly after nanokernel and basic hardware initialization), canary value variation is bound to be minimal thus reducing canary entropy. In addition, an attacker who learns a target system's TSC value at some point and who can estimate the system's startup time can carry out (depending on target clockspeed) an attack similar to the one against QNX's stack canary mechanism as outlined in Section 5.1.4. Finally, because this implementation of the random API simply outputs the lower 32 bits of the x86 TSC, every non-secret use of the random API (eg. a task generating a public random value) constitutes an *information leak* since it allows the attacker to learn (part of) the current TSC value from which the *master canary* is directly derived.
    - `TIMER_RANDOM_GENERATOR`: This option enables number generation based on the system timer clock. The random API implementation, illustrated in Listing 13, fetches

the current clock counter value and adds an (increasing) increment to it in order to ensure a series of rapid calls to the API return different values. The current clock counter fetching function `k_cycle_get_32` is implemented on a per-chip basis as part of the corresponding timer device driver. In addition to suffering from similar issues to the above `rdtsc` approach, this approach is also particularly prone to *information leaks* due to the fact that system timer values get used everywhere (eg. available as a shell command, when displaying uptime, etc.). As such in addition to any non-secret usage of the random API constituting an *information leak*, every non-secret use of `k_cycle_get_32` also constitues an *information leak*.

– `RANDOM_HAS_DRIVER`: This option to be enabled by individual random drivers to signal that there is a true random number generator driver. This option has the following *'sub-options'*:

  * `RANDOM_MCUX_RNGA`: This option enables the random number generator accelerator (RNGA) driver. By default, support is only available for NXP Kinetis K64F MCUs. The RNGA is seeded using the result of a call to the `k_cycle_get_32` system timer function. This is ill-advised given that the system timer is both a low entropy source as well as a non-secret value (see discussion above). The impact of this might be somewhat mitigated because the RNGA hardware uses two free running ring oscillators to add entropy to the seed value but given that: a) the ring oscillators are only active when the RNGA is in active mode, b) the RNGA is only set in active mode when entropy is drawn from it and c) the *master canary* is the first random value drawn from the RNGA during system boot, this might not be sufficient. Finally, rather than use RNGA output as input for a secure PRNG, it is returned directly by the RNG API for use in cryptographic and security-sensitive applications. This is despite the fact that the Kinetis K64 reference manual explicitly states that this is ill-advised in the face of potential attacks on the RNGA internal shift registers [74].

  * `RANDOM_MCUX_TRNG`: This option enables the true random number generator (TRNG) driver, based on the MCUX TRNG driver. By default, support is only available for NXP Kinetis KW41Z MCUs which feature a NIST-compliant TRNG.

System designers not working with the appropriate Kinetis MCUs (and thus not able to make use of the RNGA and TRNG drivers) thus have to either write their own hardware RNG drivers or integrate third party ones. Furthermore, as shown in Table 27, the majority of boards supported by Zephyr do not have hardware RNG support and of those that do (all of which are ARM-based) many do not have open-source driver support for the hardware RNG. In the absence of both hardware RNG and OS CSPRNG support, system designers wishing to deploy Zephyr with stack canary protection are forced to either implement their own OS CSPRNG or rely on one of the two insecure `TEST_RANDOM_GENERATOR` options.

Information Leak Issues: The fact that the *master canary* value is a system-wide value, generated once during startup and shared across Zephyr tasks makes it particularly prone to information leaks, regardless of the quality of the canary generation mechanism, since a leak in one task can be reused against another task. Furthermore, the output of the random number generation API `sys_rand32_get` is used in public fields everywhere (eg. MAC addresses for various protocols, TCP sequence numbers, ICMP echo request identifiers and sequence numbers, DNS transaction IDs, protocol timing delays, etc.) which means it is absolutely crucial that the underlying RNG is cryptographically secure.

Canary Brute-Force Susceptibility: While the nature of the Zephyr operating system means it does not support the forked and pre-forked networking server models so susceptible to canary bruteforcing attacks [354], its `__stack_chk_fail` implementation effectively invokes the system fatal error handler `_SysFatalErrorHandler` which, by default, attempts to abort the current task and allows the system to continue operating. Given that the *master canary* value is generated once at startup and never renewed, a brute-force attack might be feasible if violating tasks get restarted by the system after shutdown. Depending on the PRNG implementation (see discussion above) an exhaustive brute-force attack might be feasible but a *byte-for-byte brute-force attack* [354] is preferable here and less restricted than in the QNX case since there is no default NULL-terminator in the canary. If system designers have chosen to re-implement the `_SysFatalErrorHandler` routine in such a fashion that it reboots the system or if tasks do not get restarted after shutdown, however, brute-force attacks seem out of the question.

5.4 CONCLUSIONS

Table 13 presents an overview of the qualitative analysis of the exploit mitigations and OS CSPRNGs of three embedded operating systems outlined in this chapter. From these analyses we can observe that mitigations are often: a) *absent* (even on operating systems which support at least some mitigations), b) *incomplete* (eg. not covering all memory objects for ASLR, not implementing canary generation handlers or insecure default settings) and c) *vulnerable* (eg. insecure randomization, information leaks or correlation attacks). Furthermore, we can conclude that:

1. Operating systems not derived from a Linux-, BSD- or Windows basis tend to face exploit mitigation and dependency integration challenges leading to (potentially insecure) design idiosyncrasies. In addition, a lack of (extensive) public security scrutiny tends to translate into relatively easily exploited mitigation(-related) vulnerabilities as evidenced by the information leaks affecting QNX or the `RedactedOS` PRNG insecurities.

2. Proper (OS) CSPRNG support and integration into exploit mitigations is required to provide secure randomization as evidenced by issues affecting QNX ASLR and Stack Canaries as well as Zephyr Stack Canaries on many development boards.

3. Issues in embedded (OS) CSPRNG design tend to translate to insecure designs with issues of varying severity as evidenced by our analyses of the QNX and `RedactedOS` OS CSPRNGs.

In the course of our qualitative analysis, we responsibly disclosed the discovered issues to the vendors in question and collaborated with those drafting fixes. As a result of our disclosures, QNX has released patches for 6.6 and redesigned parts of the upcoming 7.0 release to incorporate fixes and improvements to their ESP, ASLR, Stack Canary and OS CSPRNG designs as well as other vulnerabilities we discovered but which are out of scope of this publication. In Chapters 7, 8 and 9 we will propose mitigation designs that can be incorporated in Zephyr.

Component	Support	Issues
QNX		
Userspace		
ESP	✓	Insecure Default Configuration
ASLR	✓	Insecure Randomization, Infoleaks
		Bruteforceable, Correlation Attacks
Stack Canaries	✓	Insecure Randomization, Bruteforcable
OS CSPRNG	✓	Insecure Design
Kernelspace		
ESP	✓	Insecure Default Configuration
ASLR	~[1]	Insecure Randomization, Infoleaks
Stack Canaries	✓	No Randomization
`RedactedOS`		
ESP	×	
ASLR	×	
Stack Canaries	×	
OS CSPRNG	✓	Insecure Design
Zephyr		
ESP	×	
ASLR	×	
Stack Canaries	✓	Insecure Randomization, Infoleaks
		Bruteforcable[2]
OS CSPRNG	×	

Table 13: Qualitative Exploit Mitigation Analysis Overview
[1] *Stack, heap & mmap are supported but not kernel image,* [2] *Depending on system fatal error handler implementation*

6 EMBEDDED CHALLENGES

In order to explain the gaps in embedded exploit mitigation adoption and implementation quality discussed in Chapters 4 and 5, this chapter will discuss the challenges faced by embedded systems developers. Based on this discussion we will identify a series of *open problems* in the field of embedded exploit mitigations and outline the design criteria for exploit mitigations and OS CSPRNGs for *deeply embedded systems*.

6.1 DEVELOPMENT PRACTICES & COST SENSITIVITY

Embedded systems development practices and design cultures are different [420] from those in desktop or web application development. Compared to the general purpose world, the embedded world is heavily fragmented [213]: markets for technologies are fragmented among many different vendors and suppliers and technologies themselves are fragmented into competing standards without clear market leaders and individual solutions for specific problems. For example, a typical embedded product [202] is put together as the result of a hardware vendor selling a chip, deployed with an operating system and some drivers, to an embedded systems manufacturer (the *original device manufacturer (ODM)*) who integrates it into the embedded system in question (adding some hardware peripherals, writing some software) and often resells it to a brand-name company who adds a user- or machine-to-machine interface and put it on the consumer market. This situation leads to the following issues:

1. **Lowest Common Denominator**: Vendors at the 'top' of the chain such as chip or embedded operating system vendors often cater to very diverse customers and as such are bound by the demands (in terms of capabilities, overhead and cost increases) of their most constrained customers.

2. **Fragmented Security Requirements**: No single entity oversees the entire software development lifecycle and as such, there is no coherent, single set of security requirements.

3. **Patching & Maintenance Issues**: As discussed in Section 2.3, in many cases no single entity has the ability to patch or upgrade every piece of software on a given embedded device once it's shipped.

4. **Incentive Issues**: While there might be a strong case for certain security measures when considering the end product as a whole, there is usually little incentive on part of individual vendors and manufacturers who are just a single link in a much bigger chain. Embedded systems markets are often characterized [202, 213, 420] by a heavy focus on time-to-market (earlier market introduction tends to mean deeper market penetration and hence higher potential revenue) and novel features: since embedded systems are designed for a specific purpose rather than general purpose computing, vendors often differentiate themselves on the basis of price and specific features rather than generic capabilities unrelated to the specific utility of the end product. As a result, there is little incentive for integrating security measures if these are not already present by default.

Finally, embedded systems are often very cost sensitive [420, 421], they tend to be produced in large quantities and as such even small cost increases per unit rapidly amount to large overall production cost increases. In addition, for the cheaper products a cost increase on part of a single component soon amounts to a higher percentage of total system cost making it harder to justify such a cost increase, especially with something that's often so hard to quantify as 'improved security'. Finally, any cost savings might aid in gaining a market advantage for price sensitive products. As such there usually is a preference for cheaper, simpler hardware such as chips with few features and limited room for overhead. This matters from a security perspective because it makes many hardware-based security measures infeasible (because of the associated per-unit cost increases) and means designers and implementers of embedded security measures have to deal with limited hardware capabilities and resource constraints.

6.2 RESOURCE CONSTRAINTS

It is a well-known fact that embedded systems generally face significant resource constraints [202, 420, 422, 423] since they are designed with a specialized, dedicated purpose in mind rather than to provide general purpose capabilities. As such all resources considered superfluous to this task are eliminated to reduce production cost which results in limitations on code and data memory, processing power and hardware capabilities. Embedded software, in turn is designed to be efficient and have a minimal footprint in order to meet these constraints given the limited room for overhead. Table 28 in Appendix A lists resources of several popular embedded / IoT-oriented development boards commonly used for prototyping to illustrate these resource constraints. In the context of this work we concern ourselves with four major resource constraint areas:

1. **Code Storage Size**: Code storage size constraints limit code size overhead and the introduction of additional functionality. Many embedded systems are diskless and do not have permanent storage, storing code in flash memory of a few KB or MB instead. Those systems that do have permanent storage use something like a few KB of EEPROM, usually to store configuration data only since the infrequent changing of code means it's more economical to arrange this via flashing a firmware update, or are far more limited than hard disk capacities of desktop or server systems (eg. using SD cards of a few GB).

2. **Memory Size**: Memory size constraints limit memory usage overhead and often rule out the possibility of memory-intensive computations. Embedded systems, particularly *deeply embedded systems*, often don't have external memory but rely only on a few KB or MB of on-chip internal (S)RAM. Those systems that do have external memory are often limited to anything from a few dozen MB up to one or two GB.

3. **Processing Power**: While there is a trend towards usage of more powerful 32-bit processors [13, 37, 240–242] running at clockspeeds ranging from 100 MHz to around 1 GHz (the average in 2015 being 397 MHz according to [242]) and there are plenty of embedded segments where even more serious computing power is a must, many embedded systems continue to use simpler 8- or 16-bit processors with clockspeeds ranging from 8 to 32 MHz as illustrated in Table 28. Such a lack of processing power inhibits deployment of computationally intensive security measures and certain cryptographic algorithms.

4. **Power Consumption**: Many embedded systems have serious power consumption constraints [420, 421] as a result of being battery operated, having to last months, years or indefinitely on a single battery while others might get recharged more frequently. As such, this constraint conflicts with security measures that introduce significant power consumption overhead (eg. due to being computationally intensive or requiring 'power hungry' hardware).

6.3 safety, reliability & real-time requirements

Since embedded systems are often (part of) *Cyber-Physical Systems (CPS)* they tend to have specific requirements relating to safety, reliability and real-time computation [202, 420, 423]:

1. **Safety & Reliability**: As pointed out in Section 2.3, some embedded systems have stringent safety and reliability requirements which would require certification of any security mea-

sures upon their introduction and require them to be robustly reliable (eg. maintain availability). This means, for example, that exploit mitigations for these embedded systems will have to avoid invocation of safety- and reliability-violating *'alert policies'* (such as abruptly terminating critical software upon detection of attacks).

2. **Real-Time**: Many embedded systems are subject to real-time requirements, where the system must guarantee response within specified time constraints with varying degrees of *'hardness'*, and use *real-time operating systems (RTOS)* to accommodate this. As such, security measures for those systems will need to respect those constraints. Ideally, such real-time compliant security measures are deterministic in nature and allow for accurate predictions of worst case performance. However, such requirements might inherently conflict with certain exploit mitigation designs or their dependencies.

Consider, for example, *Address Space Layout Randomization (ASLR)* and its dependency on *virtual memory*. Traditionally, the use of *virtual memory* in real-time operating systems has been avoided due to timing analysis complications [365]. *Virtual memory* poses predictability problems regarding *worst-case execution time (WCET)* analysis largely because of two issues [316, 365, 462]:

a) **Address Translation**: Mapping virtual to physical addresses is commonly done using a *translation look-aside buffer (TLB)* [265]: a memory cache that is part of the MMU and stores recent address translations. Address translation timings are unpredictable because a) not all mappings are cached in the TLB leading to cache misses requiring a subsequent page table lookup and b) the TLB is shared between different processes.

b) **Paging**: Since physical memory is shared between different processes and any physical page may be selected for replacement by the paging algorithm, predicting whether a virtual memory reference results in a page fault is hard. In addition, paging makes memory access timings dependent on TLB and cache contents increasing unpredictability. Finally, page faults may incur significant overhead rendering a system non-responsive for too long.

Various proposals for real-time compatible *virtual memory* exist, ranging from composable schemes [316] and compile-time solutions utilizing fixed page swapping points [365, 394] to dedicated MMU designs [462], but to the best of our knowledge none of these have seen adoption by popular RTOSes and we consider widespread adoption unlikely in the near future due to significant associated performance penalties or hardware cost increases.

6.4 HARDWARE & OS LIMITATIONS

As a result of the embedded cost sensitivity and resource constraints discussed above, embedded hardware and operating systems are often lacking the features upon which modern security measures depend which is illustrated in Chapter 4 with respect to exploit mitigation dependencies. In this section, we will briefly discuss the implications of these limitations for the future embedded adoption of the exploit mitigations in our baseline as well as identify some related open problems.

6.4.1 *MPUs, MMUs & Hardware ESP*

As shown in Figure 11b in Section 4.2, under half of surveyed embedded core families have *hardware ESP* support. While 32-bit processors are clearly gaining increasing traction within the embedded world and are even displacing 8- and 16-bit processors [13, 37, 240–242], smaller 8-bit processors continue to dominate a significant portion of the embedded space and some report they have seen growth in recent years as well [73]. Even though for many systems based on those smaller 8- and 16-bit processors it's quite reasonable to migrate to popular Harvard architectures (eg. AVR, 8051, PIC, etc.), many modern 32-bit processors tend to be Von Neumann and while many popular architectures in this category have hardware ESP support (eg. ARMv6+, MIPS32r3+, x86) there are others which do not. Combined with the fact that older Von Neumann processors will continue to be produced and integrated into new systems, this will leave a segment of embedded devices without hardware ESP support. This is something we need to consider in light of the fact that currently existing software ESP solutions (eg. PaX's *NOEXEC* [174]) only support a limited number of OS and architecture combinations. As such, low-overhead software ESP support for a wide range of common embedded operating systems and processor architectures is currently an open problem.

Regardless of whether ESP is implemented using hardware ESP support or via software emulation, memory protection support on part of the operating system is required. And while Figure 10 shows that the majority of embedded operating systems offer memory protection support, not all embedded hardware offers the required underlying features to allow the OS to make use of this support. Figure 11b shows only 47.1% of all surveyed core families have MMU support and only 11.8% have MPU support, which leaves 41.1% unable to accommodate memory protection. And while these figures do not take embedded market shares into account, the top embedded 32- and 16-bit microcontroller families in 2015 according to [242] (STM32, TI MSP430) are illustrative in that the both lack MMU support, the former has varying MPU support and the latter mostly lacks MPU support as well. And when it comes to 8-bit microcontrollers, MPU and MMU support is generally out of the question and many even lack external memory support in the first place. Due to cost sensitivity as discussed in Section 6.1, embedded systems manufacturers are unlikely to migrate to costlier higher end processors with MPU/MMU support mainly for security reasons and as such this leaves us with the open problem of how to deal with these MPU/MMU-less platforms which simply cannot accommodate memory protection, let alone ESP.

6.4.2 *Virtual Memory*

As discussed above, real-time requirements and lack of MMU support adversely affect embedded virtual memory adoption. While Figures 10a and 9b in Section 4.1 show that 51.2% and 44.4% of respectively all and all *non-mobile* embedded operating systems offer virtual memory support, this drops to a mere 17.1% if we eliminate the Linux-, BSD- and Windows-based ones (as shown in Figure 10c). And when it comes to the most constrained operating systems, those targeting *deeply embedded systems*, virtual memory support is absent altogether (as seen in Figure 10d). One also needs to take into account that even if an embedded OS offers virtual memory support, diskless embedded systems cannot use this to extend RAM since this would requiring swapping to disk. All these constraints are so intrinsically tied to the embedded space that it is highly unlikely that we will see universal virtual memory adoption and as such an alternative to ASLR suitable for embedded systems without virtual memory remains an open problem.

6.4.3 *Advanced Processor Features*

Many modern security measure proposals rely on advanced processor features to offset otherwise unacceptable overhead penalties. Such features range from support for trusted computing (*ARM TrustZone* [3], *Intel TXT* [266]), virtualization (*VT-x*, *AMD-V*), complication of kernel-mode exploitation (*SMEP* [197], *SMAP*), isolation of code and data regions in memory (*SGX* [264]) and pointer bounds checking (*MPX* [263]) to features utilized to support *Control-Flow Integrity (CFI)* (*CET* [108], *LBR* [107], *MPU* [453]) as well as cryptographic hardware acceleration (eg. *AES-NI*).

When it comes to embedded systems, the problem with security measures which rely on such advanced processor features is that they're only available on the newest and most high-end architectures. Even among the more high-end embedded-oriented processors such as the *Intel Atom* or the ARMv8-based *APM HeliX* and *ARM Cortex-A35*, the vast majority of these features is unsupported. In fact, on the more commonly encountered 32-bit processors such as the *Intel Quark* or *ARM Cortex-M* series none of these features are supported, let alone smaller 8- and 16-bit microcontrollers such as the *Atmel ATtiny* or *TI MSP430* series. And such advanced processor features aren't likely to be adopted by any embedded-oriented processors other than the most high-end ones anytime soon either considering the corresponding cost increase. Moreover, the sheer diversity of embedded processors means that while advanced processor features might be present in one core family they might not be present in another one, reducing the potential deployment surface of any security measures which rely on them. As such, any proposal for embedded security measures seeking widespread adoption will need to avoid relying on such advanced processor features.

6.4.4 *OS CSPRNGs*

Secure randomness plays a fundamental role in the wider security ecosystem, not only for cryptographic purposes (eg. generation of secret keys and *nonces*) but also as a dependency upon which exploit mitigations rely. As shown in Sections 5.1.3, 5.1.4 and 5.3.1, exploit mitigations which utilize insecure sources of randomness are faced with security issues. Since the design and implementation of a CSPRNG is not a trivial affair, the provision of secure randomness can be considered an important OS service and many major operating systems provide an OS CSPRNG for this reason (eg. `/dev/(u)random` on Unix-like systems or `CryptGenRandom` [143] on Windows). But as can be seen in Figure 10, OS CSPRNG support is far from universal in embedded operating systems. This is particularly visible in the non Linux-, BSD- and Windows-based operating systems (see Figure 10c) and even more so in those aimed at *deeply embedded systems* (see Figure 10d). And those operating systems that do seek to implement an OS CSPRNG, quite often make design and implementation mistakes with serious security consequences (as shown in Sections 5.1.5 and 5.2.1 as well as related work [147, 326, 407]).

Porting existing OS CSPRNG designs from the general-purpose world to the embedded world, even if it is from a GP-oriented version to an embedded-oriented version of the same operating system, is far from trivial for various reasons core among which are:

1. **OS & Hardware Polyculture**: As discussed earlier in this work, the embedded world is heavily fragmented. The fact that embedded operating systems often seek to cater to platforms with much more divergent capabilities than their general-purpose counterparts means it is hard to identify universally available, suitable entropy sources. So while there exists a sizeable body of work around the design of embedded random number generators, these designs are generally very domain-specific as they rely on entropy sources (eg. sensor values [313, 348, 384, 396, 428], gyroscope or accelerometer measurements [313], radio and GPS data [302, 384], etc.) present only in specific embedded devices.

2. **Resource Constraints**: The resource constraints discussed in Section 6.2 also impact embedded PRNG design. Limited processing power, memory and code size constraints translate to a need for *lightweight cryptography* [159]: small, fast algorithms which still offer the appropriate degree of security. In addition, power consumption constraints necessitate a PRNG design that avoids constant entropy collection, especially considering many battery-operated *deeply embedded* devices spend most of their time in standby modes waiting for event- or time-based activation to preserve battery life.

3. **Low Entropy Environment**: Perhaps the biggest hurdle in embedded PRNG design is the fact that embedded systems are generally a low entropy environment. Since they are designed for specific, limited tasks and designed to perform those in a reliable fashion, they are essentially *'boring'*. PRNGs are deterministic and effectively *'stretch out'* seed data into a stream of pseudo-random bits, they need to collect this seed data from an external source to avoid a *'chicken-and-egg problem'*. Ideally, this is done using a hardware *True Random Number Generator (TRNG)* based on physical phenomena, either *quantum-random* ones such as radioactive decay or shot noise or non-QR ones such as thermal noise, atmospheric noise or sensor values. But in practice, most systems lack such integrated TRNGs and thus have to resort to using other entropy sources. In the general purpose world, where one can assume most systems have user peripherals and disks one can use the associated system events (eg. keystroke timings, mouse movements, disk access timings, etc.) as a source of entropy. But for most embedded systems, being headless and/or diskless as well as having no user interaction, this is not an option.

These low entropy conditions are worst at boot. Boot sequences are predictable and designed to be fast, with little interaction taking place and some entropy sources not being available for entropy gathering yet, which leads to insufficient entropy being available in the PRNG pool. A big problem in embedded systems, however, is that system services need to be available rapidly after boot and as such many embedded systems with OS CSPRNGs (as shown in Sections 5.1.5 and 5.2.1 and discussed in [407]) offer only *non-blocking* interfaces which allow for drawing pseudo-random bits from the OS CSPRNG while insufficient entropy is available, leading to the so-called *'boot-time entropy hole'* [407]. Randomness generating during this *'boot-time entropy hole'* is of low quality and unsuitable for security purposes, but many embedded systems generate secrets based on such randomness precisely straight after booting. This situation, together with the general low entropy conditions in embedded systems, has led to a myriad of cryptographic vulnerabilities [133, 321, 327, 380, 407].

Ideally, this problem would be solved by having omnipresent, on-chip, high-throughput TRNGs available but considering embedded cost sensitivity issues this is not realistic. So in practice, one sees a lot of 'workarounds' of dubious quality, which tend to lead to security issues of their own. Common and insecure approaches are to use *'personalization data'* (eg. device MAC addresses or serial numbers) as seed entropy [79, 331] or rely on manufacturer-supplied initial entropy [353], sometimes in the form of a so-called *'seed file'*. Seed files are widely used in general-purpose OS CSPRNG designs in order to supplement boot-time entropy and work by having PRNG-generated pseudo-random bits written to them upon system shutdown and being drawn upon during system boot. This approach still leaves various problems for embedded systems, however, such as dealing with diskless nodes and not being applicable to the first system boot (which is often when embedded devices generated their long-term cryptographic keys). An approach sometimes encountered is to include such a seed file with device firmware but care needs to be taken here that these seed files are unique per device, unpredictable and secret. Given these problems, many embedded systems designers often simply opt for using hardcoded, pregenerated keys embedded in a firmware image with all the accompanying security issues [41, 138].

6.5 OPEN PROBLEMS

Based on the material above we can identify two pressing open problems relating to embedded exploit mitigation adoption:

1. **Mitigation Designs for Deeply Embedded Systems**
2. **OS CSPRNG Design for Deeply Embedded Systems**

We will seek to address these open problems in Chapters 7, 8 and 9.

6.5.1 *Deeply Embedded Exploit Mitigation Criteria*

We can distil the following criteria for *deeply embedded* exploit mitigations based on the observations in this chapter:

1. **Limited Resource Pressure**: Mitigations should limit pressure on constrained resources to a minimum and provide low *worst-case* (rather than *average-case*) overhead upper bounds. As observed by Szekeres et al. [390] and the rules of the Microsoft BlueHat contest [42], exploit mitigations are only likely to see widespread industry adoption if the *average-case* imposed code size, memory and runtime performance overhead is between at most 5 and 10%. As such we require a *worst-case* overhead upper bound of at most 5% for mitigations targeted at *deeply embedded systems*.

2. **Hardware Agnostic**: Mitigation designs should be hardware agnostic in order to widen deployability across the embedded hardware polyculture. This rules out any dedicated hardware proposals and any reliance on specific hardware features that aren't commonly available in deeply embedded systems. This does not include hardware features commonly but not universally available such as hardware ESP.

3. **Availability Preservation**: Mitigations should offer multiple measures to take upon detection of an attack that allow for different degrees of availability preservation, ranging from those that allow an attack to take place without interfering to those that reduce availability disruption to a minimum. The rationale behind the former is that if availability is of prime importance, the worst-case scenario for an exploited vulnerability is to disrupt this availability and as such an unhindered but reported attack that gains control of the system and keeps it up is preferable over a prevented attack that brings it down in the process.

4. **Real-Time Friendly**: Mitigations should not violate real-time requirements and as such avoid non-deterministic constructs. As discussed earlier, this rules out designs relying on *virtual memory*. Engineering mitigations for *worst-case* overhead estimates as a result of resource pressure minimization aids with *worst-case execution time (WCET) analysis* as well and thus benefits real-time friendliness.

5. **Easy (RT)OS Integration**: Mitigations should be easy to integrate into existing operating systems without requiring significant redesign of the operating system itself in order to widen deployability across the embedded OS polyculture and reduce integration cost. For example, many *deeply embedded* operating systems are implemented to be loaded as a single, monolithic image executed in a single address space without any dynamic run-time loading and as such cannot accommodate any runtime randomization measures (eg. ASLR) without significant reengineering.

6.5.2 *OS CSPRNG Design for Deeply Embedded Systems*

We can distil the following criteria for *non-domain specific deeply embedded OS CSPRNGs* based on the observations in this chapter:

1. **Lightweight Cryptography**: The CSPRNG will have to be based on lightweight cryptographic primitives [159, 204] to accommodate code & data memory as well as processing power constraints. Given the diversity of systems falling under the *deeply embedded systems* umbrella it is hard to delineate precise constraints, but any OS CSPRNG design targeting *deeply embedded systems* should be deployable on a representative hardware platform and only utilize a small fraction of available resources.

2. **Entropy Gathering Limitations**: The CSPRNG will have to be designed in such a way as to not rely on constant runtime entropy gathering in order to reduce power consumption. This means entropy collection will have to be rapid and preferably take place mostly during system startup, given that many battery-operated embedded systems are in standby or powered off between small periods of event- or time-triggered activity.

3. **Non-Domain Specific Entropy Sources**: The CSPRNG will have to draw upon entropy sources that are both suitable in terms of entropic quality as well as nearly universally present in *deeply embedded systems*. Ideally, such entropy sources have high throughputs so sufficient entropy is rapidly available at system startup and runtime entropy gathering can be limited. While there is nothing preventing CSPRNG augmentation with additional platform- or device-specific entropy sources (eg. sensor values), these should not be the primary sources nor should the choice of entropy sources be left up to the system integrator.

Part III

μARMOR DESIGN, IMPLEMENTATION & EVALUATION

7 µARMOR DESIGN

In this chapter we will propose µ`Armor` : an exploit mitigation and OS CSPRNG baseline design for *deeply embedded systems*. µ`Armor` seeks to address the relevant gap areas identified in Sections 4.3, 5.4 and 6.5 and is, to the best of our knowledge, the first *deeply embedded* solution that adheres to the criteria outlined in Sections 6.5.1 and 6.5.2.

µ`Armor` is targeted at those *deeply embedded systems* which satisfy the following conditions:

1. Feature either a (modified) Harvard architecture CPU or Von Neumann one with an MPU/MMU with hardware ESP support.
2. Run a *low-end* operating system (eg. Zephyr, FreeRTOS, TinyOS, etc.) with a single address space and without *virtual memory* support. The operating system is allowed to be multiple-stack, multi-threading and real-time capable. µ`Armor` could be modified to support *bare metal* systems with minimal engineering effort.

We will show that µ`Armor` improves significantly on the state-of-the-art in embedded binary security by raising the bar for exploitation of embedded memory corruption vulnerabilities while being adoptable on the short term without incurring prohibitive extra cost.

7.1 REPRESENTATIVE PLATFORM

We have designed and implemented µ`Armor` around a representative platform, but µ`Armor` is not intrinsically tied to any particular operating system or hardware configuration. Our representative platform in question is a deployment of the *Zephyr* [186] real-time operating system on a `TI LM3S6965` [106] microcontroller, which is based on a 50 MHz ARM Cortex-M3 outfitted with 256 kB flash, 64 kB SRAM and an MPU.

We chose *Zephyr* because it is an actively developed, open-source operating system with a permissive license aimed at resource-constrained embedded devices and is supported by the *Linux Foundation* and major chip vendors such as *Intel*, *NXP*, *Synopsys* and *Nordic* as well as explicitly committed to security.

We chose the `TI LM3S6965` microcontroller because it is supported by *Zephyr* (including via a *Qemu* profile) and is representative of a typical *deeply embedded system* with limited resources.

7.2 ATTACKER MODEL

µ`Armor` is designed for the following attacker model:

- **Control-Flow Hijacking via Memory Corruption**: µ`Armor` seeks to protect against *control-flow hijacking attacks* [390] which exploit *memory corruption vulnerabilities*. µ`Armor` does not seek to protect against *data-flow hijacking* or *data-only* attacks nor against non-memory corruption related attacks such as logical vulnerabilities.

- **Arbitrary Code Execution**: The assumed attacker goal is *arbitrary* code execution as per [412], ie. the attacker desires to be able to invoke arbitrary system functionality with arbitrary parameters.

- **Remote Attacker**: µ`Armor` assumes a *remote* attacker attempting to exploit a vulnerability over a networking protocol (eg. Ethernet, WiFi, ZigBee, Bluetooth, LoRaWAN, etc.) and considers physical access attacks (eg. sidechannel attacks, firmware extraction and replacement attacks, JTAG attacks, attacks influencing environmental conditions such as temperature, etc.) out of scope. Hence we assume the attacker does not have access to the specific firmware image of the target device but they may have access to the firmware image of another instance of the system (eg. a device they bought themselves for reverse-engineering and exploit development purposes).

As such, µ`Armor` does not seek to protect against threats outside the above attacker model such as information leaks or *Denial of Service (DoS)* attacks.

7.3 HIGH-LEVEL DESIGN

As illustrated in Figure 22, µ`Armor` incorporates three mitigations measures in order to match the functionality of the baseline outlined in Chapter 3: µ`ESP` , µ`Scramble` and µ`SSP` . In addition, it includes µ`RNG` in order to provide required OS CSPRNG support.

µ`ESP` is a *microcontroller-oriented* ESP design that relies on hardware ESP support and can be implemented as part of the operating system's memory management subsystem. µ`Scramble` is a software diversification technique that aims to complicate code-reuse attacks by randomizing code memory layout on a per-device basis at

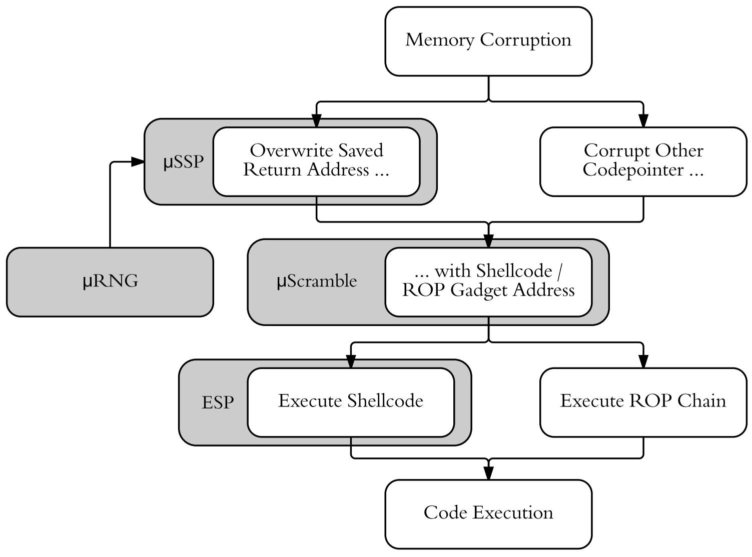

Memory Corruption
µSSP
Overwrite Saved Return Address ...
Corrupt Other Codepointer ...
µRNG
µScramble
... with Shellcode / ROP Gadget Address
ESP
Execute Shellcode
Execute ROP Chain
Code Execution

Figure 22: µ`Armor` High-Level Design as a subgraph of Figure 2

compile-time and thus seeks to provide a lightweight alternative to ASLR for systems without *virtual memory*. µ`Scramble` is implemented in the compiler and can be transparently integrated into a vendor's firmware updating mechanism. µ`SSP` is a hardened stack canary implementation for operating systems with a *single address space* and is implemented partially in the compiler and partially in the operating system (in the form of canary generation, validation and failure handlers). µ`SSP` also comes with a dedicated failure handler that allows for a degree of system availability preservation rather than the default of terminating the violating task. µ`RNG` is a small OS CSPRNG using lightweight cryptography and non-domain specific entropy sources to generate secure random numbers for use in µ`SSP` (and potentially other purposes) and is implemented in the operating system.

7.4 µESP DESIGN

µ`ESP` is the *Executable Space Protection (ESP)* component of µ`Armor` and differs from regular ESP designs in that it is explicitly designed for MCUs running single address space operating systems. µ`ESP` assumes the operating system can be modified for ESP-compliance, ie. it allows for for separating code and data memory regions as well as avoiding code constructs such as dynamically generated code, stack-stored trampolines, etc. As illustrated in Figure 23, µ`ESP` explicitly sets the hardware ESP non-executable bit for every memory region belonging to a non-code region, while not setting it for those belonging to code region. In addition, it ensures that no write permissions are set for memory regions beloging to a code region to avoid code

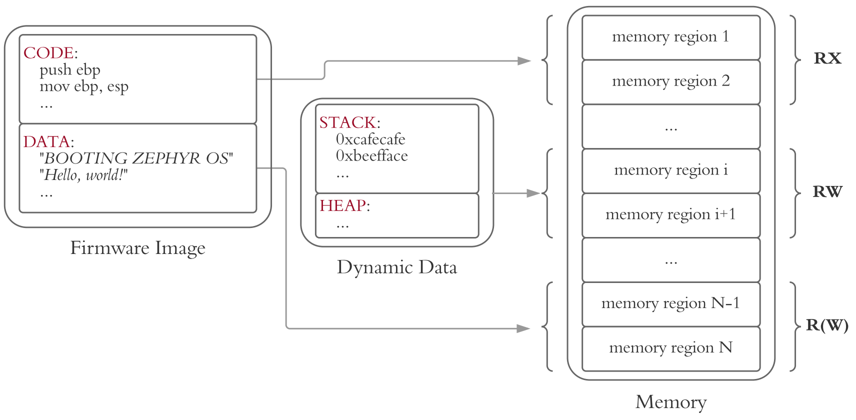


Figure 23: µESP Design

modification attacks.

Furthermore, dynamic data memory objects (eg. stack, heap) are ensured to be *fully* placed in a data memory region and the stack is instructed to grow *away* from other data regions. This is generally recommended in embedded systems [200] to prevent *stack overflows* (not to be confused with *stack buffer overflows*), resulting from unsafe embedded programming practices such as *recursion*, from corrupting data memory. Since most stacks grow downward, by placing the stack at the bottom of data memory an overflow will cause an exception either because it is a code region in RAM (thus caught by µESP permissions) or because we try to write outside of RAM. In a multi-stack environment, individual stacks overflowing into each other could be caught by placing a *guard* at the end of each stack if MPU granularity and region count allows for this.

Finally, after setting up permissions µESP will mark any code regions responsible for MPU interaction or flash rewriting (eg. in the bootloader) as non-executable to avoid code-reuse attacks targeting these regions for permission-changing payloads or *ret2bootloader* attacks [85, 288, 303] and will disable further changes to the MPU by making the relevant control registers non-writable.

Since µESP is designed for *single address space* operating systems, with no separation between individual tasks or task and kernel memory, it is either enabled on a uniform system-wide basis or disabled altogether. Allowing µESP to be enabled or disabled on a per-task basis (eg. for backwards compatibility with older ESP-violating code) would expose the entire address space to compromise. µESP can be integrated into an existing operating system and development environment by either modifying the bootloader or, if the OS in question

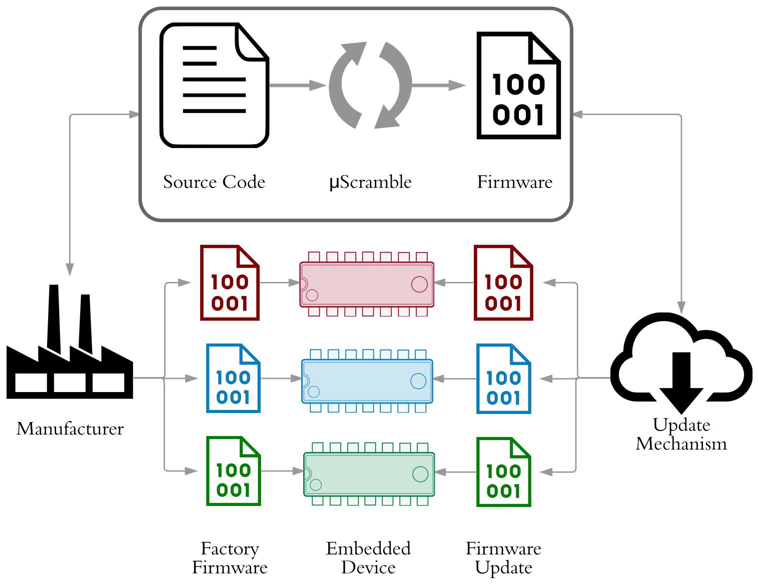


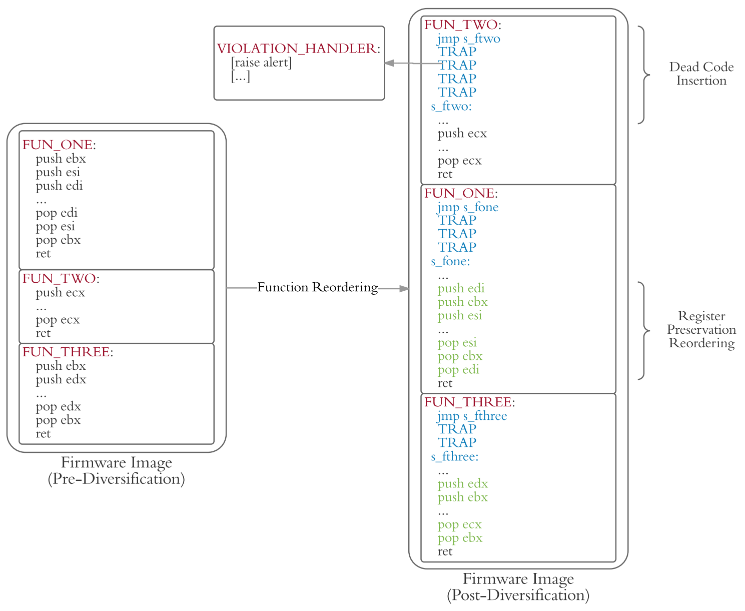
Figure 24: µ`Armor` Firmware Distribution Process

already has memory protection support, by integrating it into the memory management subsystem.

7.5 µSCRAMBLE DESIGN

As noted in Section 6.4, various limitations inherent to the embedded world inhibit widespread adoption of ASLR for code-reuse attack mitigation in deeply embedded systems. As ASLR is a runtime *software diversification* scheme which diversifies code and data memory layout, a functionally similar alternative could be drawn from the field of *software diversification* [369, 418, 447]. As pointed out by Larsen et al. [418], software diversification has two major parameters: *when* to diversify and *what* to diversify. With respect to the former, Larsen et al. identify six moments: *implementation*, *compilation & linking*, *installation*, *loading*, *execution* and *updating*. To this we could add booting (to distinguish diversification at system startup from diversification at program startup) and for many deeply embedded operating systems we could drop loading (given that small, single address space OSes don't make use of program loaders but tend to implement applications as kernel tasks).

While this field is broad and includes many diversification schemes, most of them are not suited for *deeply embedded systems*. As discussed among the related work in Chapter 10, we consider currently existing embedded *boot-time* diversification schemes such as `PASLR`, `MAVR` [371] and `AVRAND` [439] unsuited due to security flaws, requiring hardware modifications or reducing device lifespan. Since *load-* and *execution-*time solutions [418] tend to impose significant runtime overhead as



```
FUN_ONE:
    push ebx
    push esi
    push edi
    ...
    pop edi
    pop esi
    pop ebx
    ret
FUN_TWO:
    push ecx
    ...
    pop ecx
    ret
FUN_THREE:
    push ebx
    push edx
    ...
    pop edx
    pop ebx
    ret
```

```
VIOLATION_HANDLER:
    [raise alert]
    [...]
```

```
FUN_TWO:
    jmp s_ftwo
    TRAP
    TRAP
    TRAP
    TRAP
  s_ftwo:
    ...
    push ecx
    ...
    pop ecx
    ret
FUN_ONE:
    jmp s_fone
    TRAP
    TRAP
    TRAP
  s_fone:
    ...
    push edi
    push ebx
    push esi
    ...
    pop esi
    pop ebx
    pop edi
    ret
FUN_THREE:
    jmp s_fthree
    TRAP
    TRAP
  s_fthree:
    ...
    push edx
    push ebx
    ...
    pop ecx
    pop ebx
    ret
```

Function Reordering
Dead Code Insertion
Register Preservation Reordering
Firmware Image (Pre-Diversification)
Firmware Image (Post-Diversification)

Figure 25: μ`Scramble` Firmware Diversification

well as relying on either virtual memory, reliable binary analysis, dedicated hardware features or presenting problems for Harvard processors this leaves us with *implementation-*, *compile-*, *install-* and *update-*time as suitable moments of diversification.

We opted to have μ`Scramble` diversify code at compile-time because it imposes absolutely minimal runtime overhead, allows us to leverage high-level information available to the compiler, target multiple hardware platforms implicitly, avoid the need for disassembly and binary analysis as well as operate in an automatic fashion (as part of the regular compilation process) transparent to software developers (since no modifications are made to source-code). In addition, since compiler modifications should preserve program semantics, we restrict ourselves to semantics-preserving diversification transformations allowing us to preserve any guarantees on program safety which would require perfect binary analysis for non-compiler solutions. Finally, this approach is compatible with binary signing or integrity-verification mechanisms common in embedded environments.

μ`Scramble` is a hybrid **compile- and update-time solution**: since updates to program code for deeply embedded systems come in the form of firmware updates, μ`Scramble` is integrated into the firmware distribution process (see Figure 24) and involves the manufacturer compiling a *randomized* factory firmware image for *each* produced embedded device and involves the updating mechanism contacting a firmware distribution backend which compiles a *randomized* firmware update image per request. This is to ensure the μ`Scramble` diversification routine is run for each device in order to ensure *per-device* diversity which in turn ensures code memory randomization.

Next, we need to decide *what* to diversify. Since the goal of `µScramble` is to thwart code-reuse attacks, we aim to either *eliminate* gadgets, *randomize* them (to break the attacker's control- and data-flow assumptions) or make it *infeasible to guess* their location in memory. `µScramble` seeks to achieve this by diversifying code in a fine-grained manner at compile-time as illustrated in Figure 25. There exists a large body of work on *compile-time* diversification transformations [317, 346, 418, 447] but many of these are either mixed *compile-* and *run-time* solutions or their transformations are not suitable for *deeply embedded systems*: a) They need to be *semantics-preserving* b) They need to take into account embedded resource constraints regarding code size, memory usage and performance overhead. Based on these constraints we have drawn upon work in [293, 369, 455, 456] and augmented it, yielding the diversification transformations listed below for `µScramble` :

1. **Register-Preservation Reordering**: Most architectural calling conventions specify which registers are callee-saved and which are considered *'scratch'* registers. Compilers take note of all registers used within a given subroutine and will ensure that those which are callee-saved are stored to the stack during the function prolog and restored from it during the epilog. Such sequences one of the most common targets for code-reuse gadgets due to their ability to act as register-setters terminated by a return instruction. Since the exact *order* in which these registers are saved to and restored from the stack doesn't matter, however, we can randomize it and thus break gadget chain assumptions about what values end up in what registers.

   It should be noted, however, that this technique does not apply to all architectures since some (eg. ARM [4]) always push and pop registers to and from the stack in a fixed order. However, on those architectures where it does apply (eg. x86, 8051, AVR, MIPS, 68k, etc.) it significantly complicates code-reuse payload construction.

2. **Dead Code Insertion**: While the number and ordering of basic blocks within a given function can be randomized, basic block placement affects compiler optimizations [282, 315] and as such randomization would introduce potentially significant runtime overhead. As such we opted for a *dead code insertion* transformation [38, 312, 317, 372, 455, 456] which creates a random-length basic block at the bottom of a given function. The sole purpose

of this insertion is to introduce variation into function size and as such diversify the offsets of all instructions with respect to their function entry points and offsets from one function address to another. Since basic block is introduced at the bottom of a function, after the final branch instruction, it is never executed.

When it comes to selecting the content of the *dead code* we need to avoid introducing additional gadgets as part of the *dead code* basic block. We support two possible approaches here: a) We fill the block with *no-operation (NOP)*-equivalent instructions that do not present opportunities for *unaligned instruction gadgets* b) We fill the block with *trap instructions* which activate a violation policy handler upon execution (something which never happens during regular execution). The latter has the benefit of raising alerts while any attempt to brute-force gadgets is *in progress*.

Finally, since this transformation grows code size, we limit it by developer-tunable parameter DC_I indicating the maximum amount of *dead code* instructions introduced to the overall final image. Given the number of functions N in the target code, this gives us $DC_f = \frac{DC_I}{N}$ as the maximum number of *dead code* instructions introduced on any given function.

3. **Function Reordering**: Randomization of the order in which functions within a given firmware image are laid out is a common compile-time diversification technique [317, 455] which randomizes function offsets with respect to a given image base which complicates *function reuse*-style attacks (eg. *ret2lib*) in addition to more general gadget-based *code-reuse* attacks. Here we randomize only the function order and as such the degree of diversification introduced is determined by the number of functions present in the target code.

The above transformations affect both code topology and code itself by randomizing the offsets of a) instructions with respect to a function address, b) functions with respect to the image base and c) one function with respect to another function as well as randomizing the order of register preservation code. Due to the fine-grained nature of our diversification we reduce memory object *correlation* and offer better protection against information leaks and brute-force attacks than coarse-grained schemes such as base-address only randomization (eg. regular ASLR).

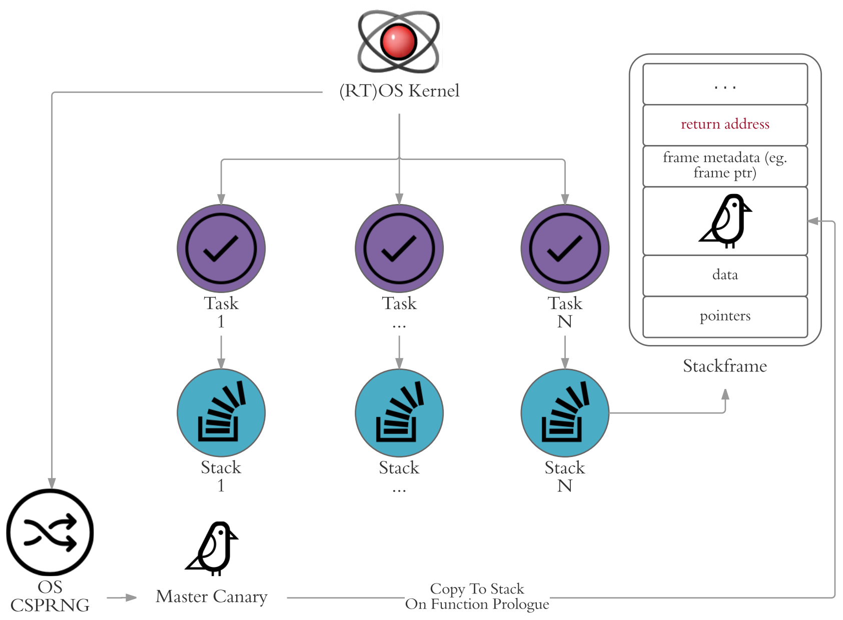


Figure 26: µSSP Design

7.6 µSSP DESIGN

`µSSP` , illustrated in Figure 26, is the stack canary component of `µArmor` and is based on *Zephyr*'s adaption of the GCC *Stack Smashing Protector (SSP)*, extended to meet the criteria outlined in Section 6.5.1. While based on the *Zephyr* SSP design, `µSSP` is not limited to any operating system or compiler in particular and is designed to be easily integrated into comparable deeply embedded (real-time) operating systems.

On the **compiler side**, `µSSP` ensures proper separation of *data* and *pointers* within a given local stackframe by placing the latter below the former so that stack overflows cannot target code- or datapointers residing on the stack while the stack canary shields the stackframe metadata (saved framepointer, return address, etc.). Note that this separation has its limitations, for instance: structures cannot be internally reordered and this could allow for stack overflows overwriting pointers. `µSSP` also complies with regular GCC SSP function coverage parameters and is capable of protecting all kernel- and application-code that runs after early kernel and C support initialization.

On the **operating system side**, `µSSP` uses a single master canary generated once at system boot for all OS tasks and threads. Since on deeply embedded systems without virtual memory there is no memory isolation for OS tasks nor a separation between kernel- and userspace, periodic canary renewal would lead to synchronization conflicts for a shared canary. A possible solution for this would be assigning a dedicated master canary for each thread (or possibly only OS tasks) as well as one for the kernel and renewing canaries upon thread startup. The problem here, however, is that without virtual memory different threads utilize shared code which would require the compiler to figure out which code is used exclusively by a given thread and which code is shared, assigning a single common master canary for all shared code to prevent synchronization problems. This limits renewal effectiveness to such a degree, especially compared to incurred overhead cost, that we simply opt for the single master canary approach. In addition to that, the single address space nature (and accompanying lack of privilege separation) of most deeply embedded OSes would render a multi-canary scheme rather moot as well.

As far as canary generation is concerned, μ`SSP` assumes the presence of either an OS CSPRNG (eg. μ`RNG` described in Section 7.7) or a TRNG. It draws a 32-bit random number from the random number generator and, if specified, it applies a bitwise and-mask of `0xFF00FFFF` to it in order to turn it into a terminator canary with 24 bits of entropy, otherwise it uses a non-terminator canary with 32 bits of entropy.

The final components of μ`SSP` is its modular canary violation handlers. As discussed in Section 6.5.1, deeply embedded exploit mitigations should offer multiple courses of action to be taken upon attack detection to allow for different degrees of availability preservation. While *Zephyr* treats canary violations as a regular fatal error, μ`SSP` seeks to offer more flexibility since fatal error handler implementations can vary from system integrator to integrator and potentially conflict with the desired canary violation behavior (to distinguish safety from security violations). We outline the following policies that can be taken upon canary violation, all of which raise an alert (which can be implemented in the form of an external alert or local event logging):

- **Passive**: The attack is allowed to continue uninterrupted after raising the alert.
- **Fatal**: The violation is treated as a regular fatal error and offloaded to the default system fatal error handler.
- **Thread Restart**: The current thread is restarted by the kernel and the system is allowed to continue running. In this manner, thread availability interruption is kept to a minimum. Note that this approach makes the canary more susceptible to *brute-force attacks*.

- **System Restart**: The system is restarted and memory is cleaned upon startup. In this manner, system availability interruption is kept to a minimum.
- **Thread Shutdown**: The current thread is terminated and the system is allowed to continue running, potentially with degraded capabilities.
- **System Shutdown**: The system is shut down as gracefully as possible.

Which measure is most suitable depends on system-specific criteria. For example, in high availability systems where no other measures are in place, a passive alert might be most suitable. The rationale behind this is that in scenarios where system availability is important and there is no replication in place or where there is a lack of fail-safe mechanisms to mitigate undefined behavior from the system, it might be preferable to allow an attacker to successfully run their exploit while system availability is maintained rather than terminate a thread or the entire system with potentially worse consequences. In other systems, eg. safety-critical ones where replication and other fail-safe mechanisms are in place the most suitable option might be a thread or system restart and in systems where security concerns win out over availability or where system degradation might have safety consequences a thread or system shutdown might be most suitable. Note that thread and system restart measures can only be implemented if the system in question supports it.

7.7 µRNG DESIGN

µRNG , as illustrated in Figure 27, is a CSPRNG designed for deeply embedded operating systems conforming to the criteria outlined in Section 6.5.2. The main purpose of µRNG in the context of this work is to serve as a dependency for exploit mitigations (such as stack canary mechanisms) but depending on chosen security strength it is perfectly suitable as a general purpose CSPRNG. While, for the sake of convenience, this work describes and implements µRNG in the context of our representative platform in Section 7.1, the µRNG design is not restricted to any operating system or platform in particular and can be easily integrated into existing (real-time) operating systems for deeply embedded systems.

µRNG is based on a compact, software-only CSPRNG design for ARM Cortex-M by Van Herrewege et al. [299] with a 128 bit security strength level and utilizes the lightweight `Keccak` [351] sponge function as a CSPRNG [350], though this can be replaced by any similarly lightweight sponge function meeting the desired security criteria. Entropy accumulation is done by Keccak's sponge *'absorbtion'*

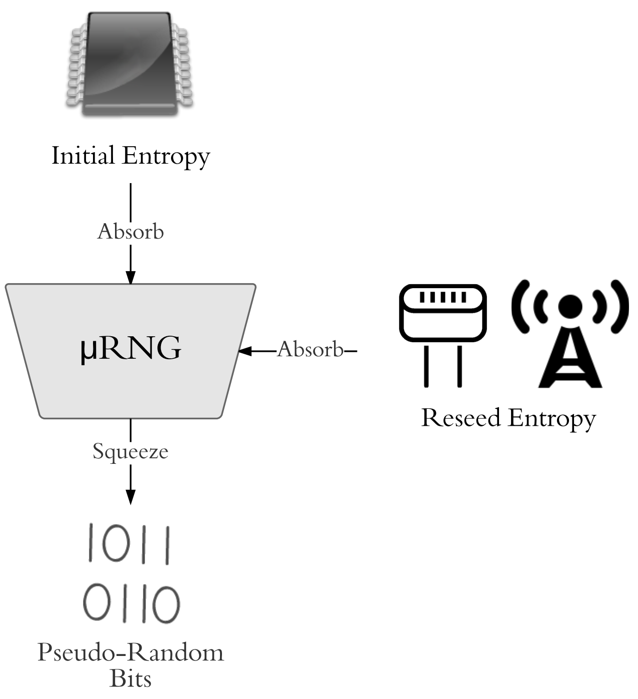


Figure 27: µRNG Design

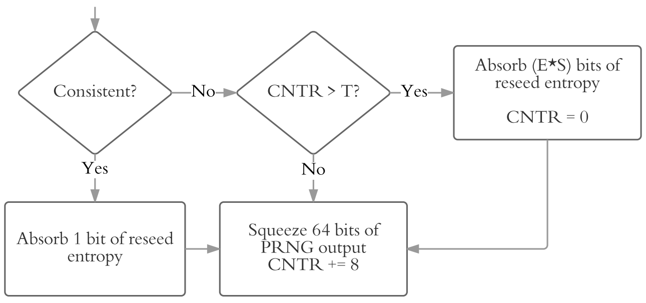

Consistent?
No
CNTR > T?
Yes
Absorb (E*S) bits of reseed entropy
CNTR = 0
Yes
No
Absorb 1 bit of reseed entropy
Squeeze 64 bits of PRNG output
CNTR += 8

Figure 28: µRNG Reseed Control

functionality while random number generation is done by *'squeezing'* the Keccak sponge, allowing us to use the same algorithm for both purposes. µRNG uses the Keccak-f[200] permutation with *rate* and *capacity* parameters r = 64 and c = 136 respectively which results in a Keccak internal state of 25 bytes and generation of 64-bit pseudo-random numbers per *'squeeze'* operation. With regards to entropy gathering, we initially seed µRNG with at least 256 bits of entropy and ensure reseeding is done with at least 256 bits of entropy as well to ensure all seeding is done with an amount of entropy that's at least double the PRNG security strength.

When designing reseed control we need to take into account the applicability of *passive* and *active* state recovery attacks [350]. In case of the former, the attacker cannot influence seed data while in case of the latter the attacker can. As per the original design by Van Herrewege et al. [299] upon which µRNG is based, µRNG provides the required security against *passive* state recovery attacks as long as reseeding occurs at least every $r * 2^{\frac{r}{2}} = 64 * 2^{32} = 32\text{GB}$ of PRNG output and against *active* attacks as long as reseeding happens at least every $2^8 * r = 256 * 64 = 2\text{KB}$ of output. Since our attacker model explicitly assumes a *remote* attacker, incapable of influencing our entropy sources remotely, we only take *passive* state recovery attacks into account. We have to consider a tradeoff between overhead and security with respect to reseed frequency: ideally reseeding is done regularly to keep as much entropy in the PRNG as possible at all times but frequent entropy gathering puts pressure on embedded resources in terms of memory and power consumption. We have opted for ensuring µRNG is fully reseeded every 1GB of output, well within desired security bounds for passive attack resistance. µRNG has two options for reseed control, as illustrated in Figure 28, to accommodate different types of systems:

- **Consistent**: Suitable for systems where a consistent invocation of minimal overhead is preferable. Reseed control here is integrated into the PRNG output function, ensuring at least 1 bit of entropy is accumulated for every 64 bits of PRNG output, thus ensuring a full 256-bit reseed every 2KB of output (incidentally also meeting active attacker criteria). The downside of this approach is that every call to PRNG output comes with a little additional overhead. If real-time guarantees are important here, system integrators need to ensure reseed entropy sources provide worst-case timing estimates on 1 bit throughput times.
- **Periodic**: Suitable for systems where a periodic invocation of a slightly more overhead-intense routine is preferable. Here reseed control is integrated into the PRNG output function as well, together with a 32-bit reseed counter which keeps track of the number of bytes output, but actual reseed functionality is only invoked after the counter exceeds a certain threshold value T. Reseed functionality is designed to run for at most S seconds (to facilitate worst-case timing estimates) and accumulate entropy while resetting the reseed counter. We determine reseed threshold value T as follows: $T = \frac{N}{2*M*E*S}$ where N is our full reseeding upper bound in bytes (ie. 1GB), M is the PRNG security strength in bits (ie. 128), E is reseed entropy throughput in bits per second and S is the reseed runtime limit in seconds. For example, with values $E = 512\text{KB/s}, S = 0.001$ this gives us $T = \frac{1024^3}{256*4194304*0.001} = 1000$ meaning µRNG would invoke reseed control every 1000 bytes of PRNG output.

In order to meet criterium 3 (Section 6.5.2), μRNG uses only non-domain specific entropy sources that can be found on most embedded devices and divides these sources into two groups as illustrated in Figure 27:

1. **Initial**: Initial entropy is gathered during early boot and should be rapidly available in sufficient quantity upon system startup to accommodate rapid system startup times while avoiding the so-called *'boot-time entropy hole'* [407].

   - **SRAM Startup Values**: We follow [299] in using *SRAM Startup values (SUVs)* [478] as our primary source of initial entropy. SRAM is a type of volatile memory where generally each cell consists of six transistors and two cross coupled inverters. The circuit formed in this fashion can assume two stable logical states forming the SUV: `0 (AB=01)` and `1 (AB=10)`. When powered up the SRAM cell state is unstable and will converge eventually to one of the two stable states. Due to manufacturing differences, the inverters tend to be slightly different which results in one being faster than the other and thus introducing a bias in this convergence resulting in a static SUV. In some cases, however, the difference is so small that the convergence is governed by thermal and shot noise, effectively rendering their SUV 'random'. While the *'static'* SRAM cells have been a subject of interest in the field of PUF security [319, 342, 403, 478], the *'random'* cells are of interest as a source of RNG entropy [96, 132, 298, 299, 319]. Using SRAM SUVs as a source of initial entropy allows us to have an entropy source that is present on most embedded devices, instantly available in (very) early boot and differs from boot session to boot session as well as (due to the *'static'* cells) from device to device even when two different devices operate under identical environmental conditions. As noted by [299], the entropy provided by SRAM SUVs is singular (ie. one can draw upon it only once) and whether it is sufficient for seeding a PRNG depends on the type of device used.

     As discussed in [96, 298, 299, 403], the amount of entropy in modern microcontroller SRAM tends to be around 5% of its total size at normal operating temperatures. This means that, on average, μRNG would require at least $\frac{2*128}{0.05*8} = 640$ bytes of SRAM to guarantee a security strength level of 128 bits, a reasonable restriction for most modern microcontrollers. The only limitation of this approach is that it cannot be applied to microcontrollers with low-entropy SRAM

SUVs (eg. the `PIC16F1825` [96]), heavy SRAM constraints (eg. less than 1 KB of available SRAM) or a combination of sensitive SRAM and unusual operating environments (very low or high temperatures reducing entropy). If one seeks to use µRNG only as an exploit mitigation dependency, lowering the security strength level to accommodate available initial entropy might be an option. Otherwise, we consider these limitations acceptable for the purposes of this work.

2. **Reseed**: Reseed entropy is gathered upon invocation of reseed control functionality and should be able to provide either at least 1 bit of entropy per invocation (in case of *consistent* reseed control) or an appropriate throughput rate (in case of *periodic* reseed control).

    - **Clock Jitter & Drift**: The various oscillators (eg. RC, Ring or VC oscillators) acting as microcontroller clock signal sources are never completely stable and are influenced by factors such as supply voltage, temperature, etc. As such, their periods tend to vary in the time (named *clock jitter*) and frequency (named *clock drift*) domains [461]: the former manifests as short-term variations of a clock's *'true period'* while the latter manifests as one clock desynchronizing with respect to another clock.

      Clock jitter has been used as a source of entropy in true random number generators [377, 460, 461] but due to measurement issues and the fact that jitter is an unwanted phenomenon which manufacturers have sought to reduce, clock drift is a more suitable source of entropy for PRNGs [177, 380, 439] for which there are multiple ways to extract entropy (eg. clock domain crossings, active comparisons of one clock against a reference clock, etc.) via software-based methods. A downside of this approach is that it relies on the existence of multiple clocks, which might make it unsuitable for the most constrained of devices.

    - **ADC Noise**: Many embedded systems are outfitted with *analog-to-digital converters (ADCs)* which have been used as PRNG entropy sources [18, 65, 305] by sampling the least significant bit of ADC output corresponding to floating inputs. While many modern microcontrollers are equipped with ADCs [160, 161], their general suitability as cryptographic entropy sources is currently unevaluated and as such we recommend against integration unless proper device-specific evaluation indicates suitability.

μRNG is by no means limited to the above entropy sources and prior work regarding embedded systems entropy has shown the suitability of less omnipresent (but still non-domain specific) sources of initial and reseed entropy: respectively *DRAM Decay* [380] and radio frequency-related sources (such as wireless transmission bit errors [302] or avalanche noise [304]). The former comes with the downside of being uncommon in most deeply embedded systems and not being instantly available like SRAM SUVs, while the latter comes with the downside of potentially being susceptible to a remote attacker influencing the RF source.

8 µARMOR IMPLEMENTATION

This chapter describes our implementation of `µArmor` for our representative platform: the *Zephyr* operating system running on a `TI LM3S6965`. While our implementation is specific to this representative platform, it is easily portable to comparable operating systems and architectures. Our implementation is meant to be a *proof-of-concept* suitable for demonstration and evaluation purposes rather than an *upstreamable patch*.

8.1 µESP IMPLEMENTATION

`µESP` can be implemented using features such as the `ARM Cortex-M3`'s optional MPU [2] (present in the `TI LM3S6965` [106]) or `Intel Quark`'s *Memory Protection Regions (MPRs)* [109]. Since *Zephyr* memory protection support will be rolled out in the upcoming Zephyr 1.8 release [163], we have chosen to implement `µESP` as a standalone component which could be integrated `µESP` into *Zephyr*'s memory management subsystem in the future.

We consider the following two common approaches to dealing with code and data memory in embedded systems: a) Program code is located in and executed from flash and data is copied to RAM and b) Both program code and data are copied from flash to RAM by a first stage bootloader and further code (eg. the operating system kernel) is executed from RAM. For `µESP` this corresponds to the permission policies outlined in Table 14. The *'sensitive'* code is that code which handles rewriting flash memory, copying data from flash to RAM and setting up memory permissions and is made non-executable after execution. The *MPU Config* refers to MPU configuration registers which are made read-only after memory permissions have been set up and the *SCB config* refers to the `System Control Block (SCB)` [90] configuration registers.

With the Cortex-M3's MPU we can enforce `µESP` 's policies by making use of its support for up to 8 memory regions. Memory regions can cover the full 4 GB address space and come with *size* (specified in bytes as a power of 2 with a minimum of 32 B) and *permission* (in the form of XN and data access flags) attributes. Memory regions start addresses must be size-aligned (ie. a 2 KB region must start at an address that is a multiple of 2 KB). We do not make use of the available privilege modes because we do not want to assume OS compatibility

Memory	**Permissions**
Code (sensitive)	RO+XN
Code (other)	RO+X
Data	RW+XN
Peripherals	RW+XN
SCB Config	RO+XN
MPU Config	RO+XN

Table 14: μESP Memory Permission Policies

with them and as such our permissions apply to both privileged and unprivileged modes. Based on the policies in table 14, we construct a μESP configuration for the TI LM3S6965 in tables 15 and 16 representing settings for *execute-from-flash* and *RAM code relocation* scenarios respectively. Listing 1 shows the pseudo-code for setting up the MPU according to μESP guidelines.

We start with a *default* region, using the lowest region number, which covers the entire address space with RW+XN permissions. If two memory regions overlap on the Cortex-M3 MPU, region attributes fall back to the region with the highest region number. We can use this feature to limit the number of regions we have to specify and define overlapping regions for exceptions to default *'data memory'*. *SRAM* and *peripherals* are covered by this default region as well. We define a region for code (covering all of flash memory) with RO+XN permissions and use a higher region as an XN overlay for any *sensitive* code (except the final line which locks the MPU) to be made non-executable after system initialization. For the scenario where code is executed from RAM, we provide identical regions to be placed wherever in RAM the bootloader relocates code to. Keep in mind that since *aliased* address

Listing 1: μESP MPU Setup Pseudo-Code

```
// mpu_set_region(id, name, perms, start, lg2size)

// setup default region
mpu_set_region(0, 'default', RW+XN, 0x00000000, 32)
// setup code region
mpu_set_region(6, 'code_other', RO+X, 0x00000000, 18)
// protect SCB by making RO
mpu_set_region(4, 'scb', RO+XN, 0xE000ED00, 6)
// make sensitive code non-executable
mpu_set_region(7, 'code_sensitive', RO+XN, *, *)
// lock MPU by making configuration area RO
mpu_set_region(5, 'mpu', RO+XN, 0xE000ED80, 6)
```

Region No.	Description	Perms.	Start Addr.	Size
0	Default	RW + XN	0x00000000	4 GB
4	SCB	RO + XN	0xE000ED00	64 B
5	MPU	RO + XN	0xE000ED80	64 B
6	Code (other)	RO + X	0x00000000	256 KB
7	Code (sensitive)	RO + XN	*	*

Table 15: `TI LM3S6965` MPU µ`ESP` Settings, *execute-from-flash*

Region No.	Description	Perms.	Start Addr.	Size
0	Default	RW + XN	0x00000000	4 GB
2	SCB	RO + XN	0xE000ED00	64 B
3	MPU	RO + XN	0xE000ED80	64 B
4	Code (other, RAM)	RO + X	*	* MB
5	Code (sensitive, RAM)	RO + XN	*	*
6	Code (other, flash)	RO + X	0x00000000	256 KB
7	Code (sensitive, flash)	RO + XN	*	*

Table 16: `TI LM3S6965` MPU µ`ESP` Settings, *execute-from-RAM*

ranges need to be covered by the same permissions, this might mean memory regions could need sizes bigger than the actual amount of on-chip SRAM.

The MPU configuration register area is located in the memory region from `0xE000ED90` to `0xE000EDBC` [106] but since this is an area of 44 bytes and the region size must be a power of 2, we will have to settle on a region size of 64. This means we also have to start the region at `0xE000ED80` because this is a multiple of 64 and `0xE000ED90` is not. Luckily the fact that this region covers more than the intended area is not an issue since there is nothing in the 'surrounding area' [270].

In addition to the above we have to consider the following security-sensitive memory regions: `Interrupt Vector Table (IVT)` and `System Control Block (SCB)`. The `IVT` holds exception vectors such as the stack pointer reset value and start address (loaded upon system reset) as well as interrupt handler addresses. The `IVT` would be an interesting overwriting target for attackers but luckily by default it lives completely within the lower region of flash memory starting at `0x00000000` and as such is covered by the `RO+X` permissions of our code region. It is possible, however, to relocate the vector table using the `Vector Table Offset Register (VTOR)` in the `System Control Block (SCB)`. If the vector table is relocated to RAM along with other code as part of a bootloader, this is not an issue because it will be covered by the relevant code region. But in order to protect an attacker from forcing a malicious relocation as part of an exploit (among other things), we mark the `SCB` as read-only. If some `SCB` functionality should be writable during runtime, `µESP` uses the MPU's *sub-region* feature which divides a region into 8 equally large sub-regions (provided the region size is at least 256 B) that can be disabled individually (thus falling back to *default* `RW+XN` permissions). If so desired, one could merge the `SCB` and `MPU` memory regions into a single region using disabled sub-regions to cover any addresses within the range which should have different permissions.

8.2 µSCRAMBLE IMPLEMENTATION

There are two popular compilers providing extensible architectures: *GCC* and *LLVM* [183]. We chose to implement `µScramble` as an extension for the *LLVM* framework, as illustrated in Figure 29, since it is more flexible and has been used extensively in prior software diversification work [293, 317, 346, 369]. While *GCC* is supported as *Zephyr*'s default compiler and *Clang* [185] (using *LLVM* as a backend) is not as thoroughly tested yet, future efforts to change this and extend *LLVM* support and testing has been announced by the *Zephyr Project* [163].

LLVM is a language- and target-agnostic compiler infrastructure that translates source-code to an *intermediate representation (IR)* and finally translates that into a machine code. This translation process consists of a sequence of transformation passes which includes functionality such as instruction selection and scheduling, register allocation and optimizations. Since *LLVM* uses a custom, abstract IR it facilitates implementing powerful new passes at different levels able to draw upon a wealth of high-level information provided by the compiler. Since *LLVM* is language agnostic, `µScramble` is as well and supports any source language.

Due to time constraints on this project, we have chosen to implement our diversification transforms only for code making proper *LLVM* passes. Given that *LLVM* passes assembly code directly to the assembler rather than through the *LLVM* pass infrastructure, our diversification does not affect code written directly in assembly. While operating systems often contain specific assembly code highly sensitive to modification, our lack of support for assembly diversification is by no means intrinsic to our design but merely to our implementation. As shown by Gionta et al. [369], diversification support for assembly code can be added to an *LLVM*-based solution by making use of the *MicroArchitectural Optimizer (MAO)* [426].

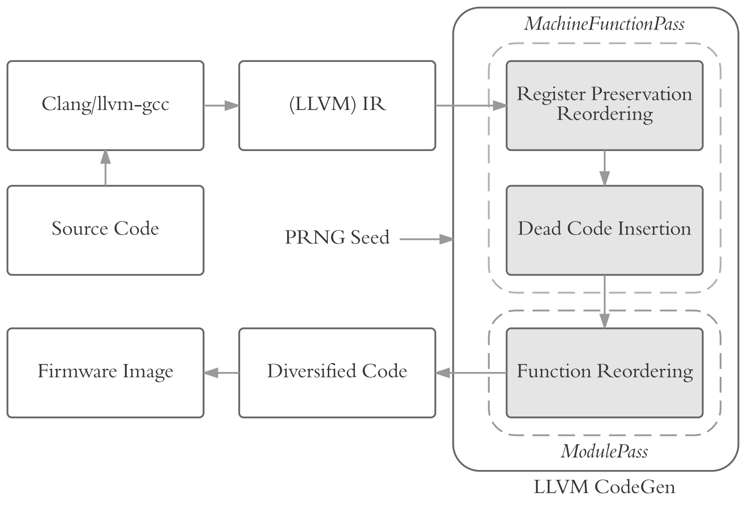


Figure 29: µ`Scramble` *LLVM* Implementation

The µ`Scramble` diversification transformations were implemented as *LLVM* passes as follows:

1. **Register-Preservation Reordering**: This transformation is implemented as a `MachineFunctionPass` [184] which obtains the callee-saved registers of a function using `getCalleeSavedRegs` [182], shuffles their order using the LLVM PRNG and sets the new order using `setCalleeSavedRegs` [182]. Note that on our representative platform this transformation has no effect since ARM (re)stores registers in fixed numerical order from and to the stack [4].

2. **Dead Code Insertion**: This transformation is implemented as a `MachineFunctionPass` which identifies the final basic block of a given function and generates a *dead code*-stub of $b \in_R B$ instructions where $B = \{n | n \in \mathbb{N}, n \leqslant DC_f\}$. It then places this stub, as a new basic block, at the end of the function. We allow developers to specify what type of *dead code*-stub (`NOP` or *trap*) they wish to generate with a compiler flag.

   `NOP`-stubs consist of a single, repeated, architecture-dependant `NOP` instruction or equivalent chosen to ensure minimal gadget usefulness if the architecture lacks a dedicated `NOP`-equivalent instruction. Our implementation currently supports `ARM` and `x86` using `andeq r0, r0, r0` (opcode: `0x00000000`) and `xchg eax, eax` (opcode: `0x90`) instructions respectively but could be trivially extended to other architectures. *Trap*-stubs consist of branch instructions to a *violation policy handler*, in our case we use the same handler used for µ`SSP` violations described below.

3. **Function Reordering**: This transformation is implemented as a `ModulePass` [184] which retrieves the current module's function list and shuffles it using the LLVM PRNG. The linker will ensure functions are organized in the randomized order in the produced firmware image.

All randomization operations used in µ`Scramble` draw upon the LLVM PRNG which draws upon a developer-supplied *true random* seed (ensured to have enough entropy to prevent frequent collisions or brute-forcing, eg. at least 128 bits). Since this PRNG is deterministic this means a given firmware build can be reproduced from the seed as is done in [369]. While the default LLVM PRNG is not secure, this does not matter much for diversification purposes because any attack on the PRNG itself would require an attacker to disclose a significant part of the firmware image thus defeating the purpose of the PRNG attack in the first place.

8.3 µSSP IMPLEMENTATION

We implemented µ`SSP` as an augmentation of *Zephyr*'s *Stack Smashing Protector (SSP)* implementation. Since *Zephyr* uses the *GCC* SSP [190] model it already meets µ`SSP` 's **compiler-side** criteria. It should be noted *Clang* supports *GCC*-style SSP as well [25]. On the **OS-side**, it stores a single master canary value as a global variable in `.bss` and initializes it at boot (as part of the `_Cstart` function, after hardware initialization but before the main thread is activated) by drawing from the `sys_rand32_get` API.

We augmented this SSP implementation by adding optional support for a *terminator-style* canary bitmask, ensuring an OS CSPRNG is available for secure canary generation (see µ`RNG` implementation below) and implementing a modular canary violation handler (commonly referred to as `__stack_chk_fail`). All violation handlers call an alert-raising function (to be implemented by the system integrator) before activating their policy measure:

- **Passive**: Here the violation handler simply returns to the violating function.
- **Fatal**: This approach, which is the default *Zephyr* approach, has the violation handler invoke the system fatal error handler with the `_NANO_ERR_STACK_CHK_FAIL` argument. By default, this will try to terminate the violating thread and continue running the system.
- **Thread Restart**: In order to properly handle thread restarts we maintain a global list (eg. a *hash table* or *association list*) of thread restart handlers associated with thread IDs. We require the thread

ID be registered together with the restart handler (to be implemented by the system integrator) upon thread start and deregistered upon thread termination. Upon invocation of the violation handler, the violating thread's ID is looked up in the restart handler list, the associated restart handler is fetched, the thread in question is terminated and the restart handler is invoked.

- **System Restart**: Here we invoke the `sys_reboot` API with the `SYS_REBOOT_COLD` argument to perform a system restart. Implementation of the API is *SoC*-specific, however.
- **Thread Shutdown**: This approach is identical to the default *Zephyr* fatal error handler.
- **System Shutdown**: This approach depends on *SoC* capabilities and power management subsystem implementations (eg. `ACPI` [257], *deep sleep* support, etc.) and as such implementation is left to the system integrator. In the absence of such functionality we default to terminating all running threads and moving into permanent *idle* mode.

8.4 µRNG IMPLEMENTATION

We implemented `µRNG` as a driver for the *Zephyr* `random` API. `µRNG` output can be requested with the `sys_rand32_get` API which *'squeezes'* the `µRNG keccak` object to produce 64 bits (the *rate* minimum) of PRNG output, the upper and lower halves of which are xor-summed together to produce a 32-bit random number as per API specifications.

Since our `µRNG` implementation uses SRAM SUVs as its initial entropy source, it is important that this entropy collection takes place as early as possible to reduce SRAM contamination (from code storing variables, using the stack, etc.) as much as possible. This means that `µRNG` initialization code should be integrated either in the system bootloader or within early kernel initialization routines. We chose to integrate `µRNG` initialization in *Zephyr*'s `__start` routine which is the firmware code entrypoint and used as the `reset handler` in the ARM Cortex-M's `vector table`. This ensures SRAM is untouched before our `µRNG` initialization code is invoked. We have not, however, used any performance or overhead-optimized `keccak` implementation. After `µRNG` has been initialized it is important to preserve the internal state (stored as a global in `.bss`) throughout the further boot process. To this end we modified the `_PrepC` routine, which sets up *Zephyr* for running C code, to ensure memory cleanup skips `KeccakState`.

9 µARMOR EVALUATION & LIMITATIONS

In this chapter we will evaluate µArmor with respect to real-time compatibility, safety issues, overhead and security against the criteria outlined in Section 6.5.1.

9.1 REAL-TIME COMPATIBILITY & SAFETY ISSUES

In order for a security solution to be real-time compatible it needs to ensure the system remains capable of meeting deadlines and remains amenable to *worst-case execution time (WCET)* analysis. Since µESP , µSSP , and µRNG are deterministic they are real-time compatibile. And while µScramble introduces a minimal degree of variability *between* different firmware images, meaning one firmware image might execute a little slower than another as a result of diversification, this does not pose a problem to real-time compatibility because: a) µScramble does not render firmware behavior non-deterministic.

In embedded systems design a tension exists between allocating too much memory to the stack and wasting a scarce resource, and too little which potentially causes stack overflows. As such stack usage is often determined *a priori* using either testing-based techniques or *stack depth analysis* [110, 364] and security solutions which modify stack usage (such as stack canaries) should take this tension into account and allow for deterministic computation of additional stack usage. As already hinted at by *Zephyr*'s inclusion of stack canaries, µSSP 's modification of stack usage is amenable to both analysis techniques in a multi-stack environment since it is deterministic: it introduces an additional 32-bit word for every function call. In addition, µESP 's stack placement is designed to have the stack grow away from other data regions in order to limit the safety-related impact of any stack overflows as is recommended [200].

9.2 OVERHEAD EVALUATION

We evaluate the overhead imposed by µArmor in terms of code size, data size, memory usage and runtime increases. Due to time and measurement equipment limitations, we consider energy consumption overhead out of scope for this work. Code and data size figures represent increases in code (in flash) and constants (in SRAM) respectively. Memory usage increases represents a worst-case SRAM overhead imposition (by use of dynamic data structures) at any point during execution. We instrumented application code to measure runtime performance using a hardware high precision counter. Runtime performance figures represent increases in the number of clockcycles consumed for a given amount of code to run, reported as the average of 25 runs.

We evaluate μ`ESP` , μ`SSP` , μ`RNG` separately in Tables 17, 18 and 19 and μ`Scramble` in Tables 20, 21, 22 and 23. In order to get an idea of the overhead on realistic applications we chose three sample *Zephyr* IoT applications stressing different subsystems:

- `philosophers` [274]: An implementation of the dining philosophers problem using multiple preemptible and cooperative threads of differing priorities.
- `net/echo_server` [275]: An IPv4/IPv6 UDP/TCP echo server application.
- `net/telnet` [276]: An IPv4/IPv6 telnet service providing a shell with two shell modules: `net` and `kernel`.

We evaluate μ`ESP` , μ`SSP` and μ`RNG` against the above representative applications compiled for the `TI LM3S6965` with *GCC* as provided by the *Zephyr SDK*. We evaluate μ`Scramble` against a different set of applications, however, since the overhead imposed by μ`Scramble` on a single application is non-deterministic and scales strongly with respect to parameters such as the number of functions. As such we chose to evaluate μ`Scramble` against a set of 50 benchmarks and applications from the *TACLeBench* suite [357] as listed in Tables 29 and 30. *TACLeBench* is designed for *Worst-Case Execution Time (WCET)* analysis and consists of self-contained programs without external or OS dependencies drawn from well-known (embedded) benchmarking suites such as *DSPStone* [39], *MRTC WCET* [141], *SNU-RT* [134], *MiBench* [399], *MediaBench* [40], *NetBench* and *HPEC* [135]. The benchmarks in question are drawn from various embedded domains ranging from automotive and networking to security and telecommunications and are sub-divided into *application*, *kernel*, *sequential* and *test* groups implementing realistic applications, small kernel functions, large sequential functions and artificial stress tests respectively.

We are not interested in average memory usage overheads but rather in worst case figures because of potentially unacceptable SRAM pressure. Using *stack depth analysis* [110, 364] embedded developers get an indication of the maximum amount of memory used by the stack in their application. Obtaining an absolute stack depth upper bound is done using static analysis but is complicated by various factors such as recursion, indirect function calls, input-dependant code paths, loops without explicit limits and multi-threading and requires

Application	% Code Size	% Data Size	% Memory[1]	% Runtime[2]
Wrt. application				
`philosophers`	1.2	0	× (0 B)	0
`net/echo_server`	0.2	0	× (0 B)	0
`net/telnet`	0.2	0	× (0 B)	0
Wrt. resources				
`philosophers`	0	0	0	×
`net/echo_server`	0	0	0	×
`net/telnet`	0	0	0	×

Table 17: µ`ESP` Overhead Evaluation
[1] *Worst-Case Estimate*, [2] *Average of 25 runs*

proprietary tools. As such we have decided to obtain a *stack depth estimate* in between the usual lower bounds derived from experimental observation and the upper bounds derived from static analysis. This estimate is derived from multiplying the longest identified *call chain* in the program *Control Flow Graph (CFG)* by the overhead imposed by a single canary. Ideally, we would obtain such an estimate using source-code analysis but given *Zephyr's* mix of C and assembly source-code files we chose to write a custom *IDAPython* [99] script for the *IDA Pro* [181] binary analysis framework that would perform this analysis. While limited by the inherent incompleteness of binary analysis, we believe the impact of this on simple IoT applications such as the case studies below to be negligible.

Note we report overhead figures both with respect to the original unprotected application and with respect to total device resources since the former represents the resource consumption increase µ`Armor` components impose upon an individual application while the latter more accurately represents actual resource pressure. Exceptions to this are memory usage and runtime overhead figures. The former are reported in absolute terms only since they express a worst-case scenario with respect to total device resources and the latter can be reported with respect to the original application only. We round overhead to 1 decimal place.

Based on the reported figures above, we can conclude **code size** overheads stay below 5% with respect to the application for all components except µ`SSP` and µ`RNG` and are less than or equal to 5% with respect to total device resources for all components. **Data size**, **memory usage** and **runtime** overheads all stay well below 1% both with respect to the application as well as with respect to total device resources. µ`SSP` **code size** overheads are clearly the heaviest overhead imposition of all metrics and components. µ`SSP` introduces roughly 4

Application	% Code Size	% Data Size	% Memory[1]	% Runtime[2]
Wrt. application				
`philosophers`	30.5	0	× (48 B)	0
`net/echo_server`	26.4	0	× (84 B)	0.7
`net/telnet`	27.3	0	× (84 B)	0.7
Wrt. resources				
`philosophers`	0.9	0	0	×
`net/echo_server`	5	0	0	×
`net/telnet`	5	0	0	×

Table 18: µ`SSP` Overhead Evaluation
[1] *Worst-Case Estimate,* [2] *Average of 25 runs*

Application	% Code Size	% Data Size	% Memory[1]	% Runtime[2]
Wrt. application				
`philosophers`	10.2	0.4	× (52 B)	0
`net/echo_server`	1.4	0.1	× (52 B)	0
`net/telnet`	1.5	0.1	× (52 B)	0
Wrt. resources				
`philosophers`	0.3	0	0	×
`net/echo_server`	0.3	0	0	×
`net/telnet`	0.3	0	0	×

Table 19: µ`RNG` Overhead Evaluation
[1] *Worst-Case Estimate,* [2] *Average of 25 runs*

Application	% Code Size[1]	% Data Size[2]	% Memory[3]	% Runtime[4]
Application				
lift	1.5	0	0	0
powerwindow	2.2	0	0	0
Kernel				
binarysearch	3.8	0	0	0
bitcount	2.3	0	0	0
bitonic	4.2	0	0	0
bsort	3.5	0	0	0
complex_updates	1.6	0	0	0
countnegative	3.4	0	0	0
fac	4	0	0	0
fft	2.6	0	0	0
filterbank	1	0	0	0
fir2dim	1.5	0	0	0
iir	2.2	0	0	0
insertsort	1.9	0	0	0
jfdctint	1.3	0	0	0
lms	1.4	0	0	0
ludcmp	0.9	0	0	0
matrix1	3	0	0	0
md5	1.9	0	0	0
minver	0.8	0	0	0
pm	0.8	0	0	0
prime	1.8	0	0	0
quicksort	0.9	0	0	0
recursion	4.6	0	0	0
sha	1.8	0	0	0
st	2	0	0	0
basicmath	0.7	0	0	0

Table 20: μScramble Overhead wrt. Application
[1] *Average of 25 variants,* [2] *Average of 25 variants,* [3] *Average of 25 runs of 25 variants,* [4] *Average of 25 runs of 25 variants*

Application	% Code Size[1]	% Data Size[2]	% Memory[3]	% Runtime[4]
Sequential				
adpcm_dec	1.1	0	0	0
adpcm_enc	1.3	0	0	0
ammunition	1.1	0	0	0
anagram	1.8	0	0	0
audiobeam	1.6	0	0	0
cjpeg_transupp	1.2	0	0	0
cjpeg_wrbmp	2.1	0	0	0
dijkstra	1.9	0	0	0
epic	1.6	0	0	0
fmref	1	0	0	0
gsm_dec	1.3	0	0	0
h264_dec	0.8	0	0	0
huff_dec	1.8	0	0	0
huff_enc	1.6	0	0	0
mpeg2	1.1	0	0	0
ndes	1	0	0	0
petrinet	0.2	0	0	0
rijndael_dec	0.3	0	0	0
rijndael_enc	0.3	0	0	0
statemate	0.5	0	0	0
Test				
cover	1.2	0	0	0
duff	3	0	0	0
test3	0.4	0	0	0

Table 21: μScramble Overhead wrt. Application
[1] *Average of 25 variants,* [2] *Average of 25 variants,* [3] *Average of 25 runs of 25 variants,* [4] *Average of 25 runs of 25 variants*

Application	% Code Size[1]	% Data Size[2]	% Memory[3]	% Runtime[4]
Application				
lift	0	0	0	0
powerwindow	0.1	0	0	0
Kernel				
binarysearch	0	0	0	0
bitcount	0	0	0	0
bitonic	0	0	0	0
bsort	0	0	0	0
complex_updates	0	0	0	0
countnegative	0	0	0	0
fac	0	0	0	0
fft	0	0	0	0
filterbank	0	0	0	0
fir2dim	0	0	0	0
iir	0	0	0	0
insertsort	0	0	0	0
jfdctint	0	0	0	0
lms	0	0	0	0
ludcmp	0	0	0	0
matrix1	0	0	0	0
md5	0	0	0	0
minver	0	0	0	0
pm	0	0	0	0
prime	0	0	0	0
quicksort	0	0	0	0
recursion	0	0	0	0
sha	0	0	0	0
st	0	0	0	0
basicmath	0	0	0	0

Table 22: µScramble Overhead wrt. Resources
[1] *Average of 25 variants*, [2] *Average of 25 variants*, [3] *Average of 25 runs of 25 variants*, [4] *Average of 25 runs of 25 variants*

Application	**% Code Size**[1]	**% Data Size**[2]	**% Memory**[3]	**% Runtime**[4]
Sequential				
`adpcm_dec`	0	0	0	0
`adpcm_enc`	0	0	0	0
`ammunition`	0.1	0	0	0
`anagram`	0	0	0	0
`audiobeam`	0	0	0	0
`cjpeg_transupp`	0	0	0	0
`cjpeg_wrbmp`	0	0	0	0
`dijkstra`	0	0	0	0
`epic`	0	0	0	0
`fmref`	0	0	0	0
`gsm_dec`	0	0	0	0
`h264_dec`	0	0	0	0
`huff_dec`	0	0	0	0
`huff_enc`	0	0	0	0
`mpeg2`	0.1	0	0	0
`ndes`	0	0	0	0
`petrinet`	0	0	0	0
`rijndael_dec`	0	0	0	0
`rijndael_enc`	0	0	0	0
`statemate`	0	0	0	0
Test				
`cover`	0	0	0	0
`duff`	0	0	0	0
`test3`	0.1	0	0	0

Table 23: µ`Scramble` Overhead wrt. Resources
[1] *Average of 25 variants,* [2] *Average of 25 variants,* [3] *Average of 25 runs of 25 variants,* [4] *Average of 25 runs of 25 variants*

RNG	ROM	RAM
µRNG [1]	709	81
TinyRNG [302][2]	11086	471
TinyKey [428][3]	5990	790

Table 24: µRNG Resource Consumption Comparison
[1] *for TI LM3S6965*, [2] *for MICA2*, [3] *for TMote Sky*

instructions of prolog and 5 instructions of epilog overhead amounting to 36 bytes per protected function. While the average overhead with respect to the unprotected application is 28.1% we can see overhead in terms of total resource pressure remains equal to or below 5%. As such we consider our overhead criteria met.

Furthermore, it should be noted that while our µRNG implementation has not been optimized in any way it still compares favorably against existing IoT-oriented RNG implementations by virtue of its lightweight design as illustrated in Table 24 (here RAM is taken to include both data size and memory usage increases).

9.3 SECURITY EVALUATION

9.3.1 *µESP Security*

µESP protects against both code injection and code modification by enforcing a separation between code and data memory, forcing an attacker to use a code-reuse payload. By 'locking' the MPU and rendering µESP and bootloader code non-executable after it has been run, µESP protects against code-reuse attacks that seek to circumvent µESP by means of permission-changing payloads or *ret2bootloader* attacks [85, 288, 303] that seek to rewrite flash memory with attacker-injected code. This approach does not protect against *ret2bootloader* attacks against chips with a bootloader stored in *mask ROM*, however, and as such we reiterate advice from prior work [85, 457] against such bootloaders.

9.3.2 *µScramble Security*

Contrary to µESP , µScramble offers *probablistic* security and we will evaluate it with respect to the following aspects: *entropic quality* and *relocation frequency* (which determine *bruteforce susceptibility*) as well as *information leak susceptibility*. When we consider the **entropic quality** of diversification schemes on deeply embedded devices we are immediately confronted with the limitations imposed by the size of the address space which tends to range between 16 and 32 bits. In addition, code is often restricted to a limited subregion of that address space (eg. ROM only or sometimes part of RAM). The `TI LM3S6965`, our representative MCU, has 256 KB of on-chip flash which translates to an 18 bit code address space but more constrained devices (such as some of those listed in Table 28) have to make do with 8 or 32 KB of flash which translates to 13 or 15 bit address spaces respectively. On top of this, a given piece of code is not equally likely to end up anywhere within this region because of a) randomization granularity and b) areas that are tied to special functionality such as designated bootloader regions. As such the real *entropic quality* of a given randomized code address on deeply embedded systems is even less than what the already limited address space would allow for.

Despite this inherent limitation, one needs to keep in mind that the reason we employ software diversification here is to prevent code-reuse attacks as a complement to µ`ESP` 's prevention of code injection and modification attacks. The degree to which this approach is successful depends on the feasibility of an attacker being able to construct a useful code-reuse payload. Assuming an attacker is able to discover enough gadgets to construct one in the first place, deploying such a payload under software diversification conditions (eg. ASLR, µ`Scramble` , etc.) can be done either through use of information leaks (as discussed below) or through bruteforcing gadget addresses. Being a *compile-time diversification scheme*, µ`Scramble` 's **relocation frequency** is limited to firmware updates and as such isn't re-randomized for prolonged periods of time. While this certainly reduces attack complexity, µ`Scramble` 's randomization is much more fine-grained than regular ASLR and as such more resilient because reduced correlation means disclosure of a single code address does not disclose the entire memory map.

In order to get an idea of the **entropic quality** of µ`Scramble` 's transformations we performed a **coverage analysis** consisting of taking our selection from the *TACLeBench* suite and generating 1000 different µ`Scramble` -diversified variants for each benchmark. We then proceeded to harvest all gadgets from each variant using a ROP gadget harvester [199] and determined, for each gadget in each variant, in how many other variants the gadget still resides at the same address. We then obtained the average and maximum gadget 'survival' rates. These gadget survival rates give us an indication as to the quality of µ`Scramble` 's *'coverage'* of the target gadget space and the amount of work required on part of an attacker to scale their exploits. The results of this analysis are reported in Tables 25 and 26.

From Tables 25 and 26 we can see that highest average gadget survival rate is 101.5 (for `insertsort`) and the highest maximum gad-

Set	Avg. GS[1]	Max. GS[1]
Application		
lift	2	9
powerwindow	1.2	10
Kernel		
binarysearch	1.4	13
bitcount	63.3	180
bitonic	85.15	166
bsort	58.5	171
complex_updates	68.1	199
countnegative	2	11
fac	2.5	4
fft	1.2	5
filterbank	59.9	209
fir2dim	70.5	210
iir	2.5	8
insertsort	101.5	202
jfdctint	97	193
lms	64.2	193
ludcmp	55.4	166
matrix1	68.8	203
md5	0.6	7
minver	1.3	5
pm	12.5	97
prime	0.9	5
quicksort	1.5	9
recursion	91.6	188
sha	2.3	6
st	0.8	3
basicmath	1	4

Table 25: µScramble Coverage Analysis
[1] *Gadget Survival*

Set	Avg. GS[1]	Max. GS[1]
Sequential		
`adpcm_dec`	0.5	4
`adpcm_enc`	0.6	5
`ammunition`	3.6	147
`anagram`	0.6	4
`audiobeam`	4	79
`cjpeg_transupp`	32	96
`cjpeg_wrbmp`	1.4	3
`dijkstra`	1.3	5
`epic`	0	0
`fmref`	0.5	3
`gsm_dec`	1.3	43
`h264_dec`	100	181
`huff_dec`	18.8	91
`huff_enc`	5.8	55
`mpeg2`	0	0
`ndes`	0.6	2
`petrinet`	2	2
`rijndael_dec`	2.5	16
`rijndael_enc`	3	11
`statemate`	16.4	97
Test		
`cover`	29.8	143
`duff`	1.6	7
`test3`	0	0

Table 26: μ`Scramble` Coverage Analysis
[1] *Gadget Survival*

get survival rate is 210 (for `fir2dim`) while most remain well below those numbers. This means that in a worst case scenario, a single gadget survives across roughly 10.15% of variants on average and across 21% of them at most, requiring brute-force for all other variants. Thus an attacker constructing a code-reuse payload from a given firmware variant cannot expect any gadget in this payload to work beyond at best 10.15% of target devices and needs to keep in mind that the devices for which one gadget might work, another will not (ie. the gadget survival variants of different gadgets do not necessarily intersect). As such, prospects for *gadget chain* survival are even worse and brute-force search space scales with respect to payload length as well. Furthermore, since `µScramble` diversification operates with respect to the number of functions and the available space for dead code insertion, we expect gadget survival rates to go down for increasingly complicated applications on less constrained platforms.

While `µArmor` offers no explicit protection against **information leaks**, their impact is heavily reduced compared to other diversification solutions such as ASLR due to the fine-grained nature of `µArmor` 's diversification: ie. since function positions are decorrelated due to order and size randomization, leakage of one code address does not give away others outside of a single function scope. In order to limit the impact of information leaks even further, future work could explore the applicability of *register allocation randomization* and *call site lifting* transformations [369] or techniques like *Execute-only Memory (XoM) / Execute-no-Read (XnR)* [445, 446] and *code-pointer hiding (CPH)* [383] to *deeply embedded systems*. Finally, we observe that `µArmor` does not offer protection against code-reuse attacks targeting functionality residing at inherently fixed addresses (eg. *interrupt vector table* entries) [85, 288].

Based on our *coverage* analysis above we consider the security offered by `µArmor` sufficient with respect to our attacker model but do note that code-reuse attacks incorporating platform- or application-specific fixed addresses remain unaddressed. We leave this issue to future work.

9.3.3 *µSSP Security*

The security offered by `µSSP` is inherently constrained by the limits of stack canaries: they only protect against stack buffer overflows targeting *stackframe metadata* and not against other types memory corruption vulnerabilities nor against stack buffer overflows targeting local code- or data pointers. That being said, `µSSP` draws upon an OS CSPRNG (provided by `µArmor` in the form of `µRNG`) to generate canaries with 32- or 24 bits of entropy (depending on whether they are configured to be *terminator style* or not) which are, as such, not susceptible to either the insecure randomness issues or the system-side information leaks affecting the original *Zephyr* SSP canaries in the absence of a TRNG (see Section 5.3.1). Since the master canary value is refreshed upon system boot, the *system restart*, *thread shutdown* and *system shutdown* violation policies are not susceptible to bruteforce approaches at all. The *thread restart* policy is susceptible to bruteforce attacks, the *fatal* policy's susceptibility depends on system integrator implementation and the *passive* policy naturally offers no attack prevention security. All approaches, however, raise at least an alert upon invocation. We believe that given our attacker model (considering *remote* attackers only), 32 bits of entropy is sufficient for a bruteforce attack to be infeasible. With regards to information leaks, since a single master canary is shared among all threads this means that a sufficiently powerful information leak in any of the threads or kernel affects the entire OS. We consider this acceptible given the overhead and renewal issues otherwise incurred and general absence of privilege separation in most deeply embedded OSes.

9.3.4 *µRNG Security*

Finally, we will consider the security of µRNG . µRNG is based on the Keccak [351] sponge function which has withstood professional cryptanalytic scrutiny as a CSPRNG [334, 350, 352], offers 128 bits of security and is (re)seeded with at least 256 bits of entropy. Initial entropy is immediately available from *SRAM Startup Values (SUVs)* and as such µRNG does not suffer from the so-called *boot-time entropy hole*. As discussed in Section 7.7 µRNG reseeds every 1 GB of PRNG output, well within 32 GB bound for *passive* state recovery attacks. And while out of scope for our attacker model, µRNG reseed control could be trivially modified to meet the *active* state recovery protection 2 KB bound provided the reseed entropy source has sufficient throughput. If system integrators decide to drop reseed control (eg. due to overhead constraints), µRNG can securely produce up to 32 GB of output before 'locking' and requiring a system restart.

For µRNG to function securely it is important sufficient initial entropy is available. And while prior work [96, 298, 299, 403] has observed the amount of entropy in modern microcontroller SRAM tends to be around 5% of total SRAM size at normal operating temperatures, the suitability of on-chip SRAM still varies from device to device and hence conducting adequate measurements before deciding on SRAM SUV-based PRNG designs is important [96]. In [96] the authors provide an overview of SRAM suitability for PUF and PRNG purposes of four different microcontrollers: STM32F100R8, ATmega328P, MSP430F5308 and PIC16F1825. All of these except the last one (with a worst-case min-entropy of 1.2% corresponding to 100 bits of entropy) were found suitable for PRNG purposes since those with low intra-device min-entropy percentages (ranging from worst cases of 2% to 5.25%) tended to make up for it by slightly larger SRAM (thus corresponding to sufficiently available entropy in bits).

While we considered evaluating intra-device min-entropy for our representative microcontroller to be out of scope for this work, we believe we can reasonably assume a very conservative lower bound of 1%. We also need to consider the fact that in reality, SRAM will be contaminated somewhat during SUV accumulation and that entropy is not uniformly distributed upon collection. If we make no further assumptions about SRAM entropy distribution, we assume the worst-case scenario where every 'contaminated' SRAM byte removes a full 8 bits of entropy. In order to limit this contamination one could design `keccak_absorb` to avoid touching SRAM by avoiding stack variables, using in-place operations only and using branches that do not touch the stack (eg. jumps) thus reducing SRAM contamination to the `Keccak-f[200]` internal state of 25 bytes only. But even assuming an un-optimized implementation such as the one in [299] upon which µ`RNG` is based, SRAM contamination is limited to 52 bytes. Now consider the following formula [96] for evaluating whether we meet the desired security bounds:

$$2 * T \leqslant 8 * (S_{size} * E - S_{usage})$$

where T is desired security strength in bits, S_{size} and S_{usage} are respectively microcontroller SRAM size and SRAM contamination in bytes and E is intra-device min-entropy percentage. For our conservative estimates on the `TI LM3S6965` this yields:

$$2 * 128 \leqslant 8 * (2^{16} * 0.01 - 52)$$

which holds just fine. This is, of course, merely a very conservative estimate and as such neither a substitute for an actual measurement of the `TI LM3S6965` specifically nor popular modern microcontroller SRAM in general. We leave such a thorough evaluation to future work.

Finally, we need to consider memory retention and initial SRAM state exposure. SRAM retention effects have been shown [375] to be minimal at normal ambient temperatures but some microcontrollers show clear loss of min-entropy correlated to colder ambient temperatures [96] which again emphasizes the need for adequate individual microcontroller evaluation. Care should also be taken to ensure full SRAM resets between power cycles by grounding positive supply lines upon shutdown [96] to avoid retention. Our `µRNG` implementation takes care of initial SRAM state exposure by being integrated into *Zephyr*'s `__start` routine which is closely followed by zero'ing of the SRAM.

9.4 LIMITATIONS

First of all, `µArmor` requires either a *(modified) Harvard* or *Von Neumann* CPU featuring an MPU/MMU with hardware ESP support. We believe that this limitation is within reasonable bounds for our system to be considered *'hardware agnostic'*, since it only excludes older Von Neumann architectures for which there is currently no way to enforce low-overhead ESP.

Secondly, `µArmor` only protects against *control-flow hijacking memory corruption vulnerabilities* carried out by *remote attackers*: it doesn't protect against other types of vulnerabilities and doesn't protect against attackers with physical access. In addition, the `µScramble` component only diversifies *code memory* and only provides *per-device diversity*: it doesn't diversify *data memory* nor does it diversify on a *per-boot* or *per-application run* basis.

Finally, the `µRNG` component requires on-chip SRAM to have suitable SRAM SUV entropy (ideally evaluated on a per-chip model basis) or otherwise requires an alternative source of initial entropy meeting criteria 2 and 3 outlined in Section 6.5.2. As discussed in Section 9.3.4, prior work seems to suggest most modern microcontroller SRAM is suitable as a source for PRNG entropy.

Part IV

CONCLUSION

10 RELATED WORK

10.1 EMBEDDED MITIGATION SUPPORT & QUALITY

To the best of our knowledge no quantitative evaluations for embedded OS exploit mitigation and dependency support exists prior to this work's Chapter 4. There is a large body of work dealing with memory corruption on embedded systems but this work is generally limited to either Linux-, Windows- and BSD-based systems or bare metal systems. The body of work dealing with RTOS binary security is far smaller and includes work on QNX [9, 179, 248, 285, 322, 466, 467], VxWorks [1, 67, 153, 324], Cisco IOS [136, 140, 296] and ThreadX [247, 307]. None of this work, however, deals with the quality of RTOS exploit mitigation or OS CSPRNG implementations (if they are present at all) and as such we believe the evaluation in Chapter 5 to be the first.

10.2 EMBEDDED MITIGATION DESIGN

A large body of work exists on memory corruption and exploit mitigations, an excellent overview of which is provided in [390, 459]. Most of that work, however, deals with general purpose systems. As such we will restrict the discussion in this section to exploit mitigations specifically designed for embedded systems, *deeply embedded systems* in particular.

Address Space Layout Randomization (ASLR) is a well-established exploit mitigation technique in the general purpose world and has seen adoption in parts of the embedded world as well. Particularly mobile operating systems such as Android [124, 148, 172, 306, 360, 378, 404] and iOS [35, 58, 146, 280, 307, 387, 398] have a long history of ASLR adoption, scrutiny and subsequent improvements. This level of attention is, however, largely limited to mobile operating systems and where *deeply embedded systems* are concerned there are only three ASLR proposals: `PASLR`, `MAVR` and `AVRAND`. *Physical/Pseudo Address Space Layout Randomization (PASLR)* [210] is an exploit mitigation for the *Nintendo 3DS* game console that randomizes code memory layout by shuffling chunks of memory around in physical memory. As evidenced by exploits [187, 211] from the *homebrew scene* [261], `PASLR` was poorly designed and could be trivially bypassed and as such is not suitable.

MAVR [371] is a boottime diversification scheme, developed for *Ardupilot Mega 2.5 (APM) UAV control systems*, that randomizes binary code at a per-function granularity level. MAVR introduces specialized hardware modifications (in the form of an additional *master processor* and external flash chip) to the platform and works by having the *master processor* randomize function chunks in the original *application processor* firmware and reflashing the latter during boot-time after every N restarts or after a crash. This approach does not meet our criteria since it requires (costly) hardware modifications and reduces device lifespan (proportional to the number of restarts) by reflashing firmware every randomization since the number of write/erase cycles of flash memory is inherently limited. An attacker could even weaponize this drawback by forcing a large number of system restarts (by triggering a crash or invoking a watchdog reset) which would rapidly reduce device lifespan.

AVRAND [439] is a boottime diversification scheme, developed for the *Arduino Yun* platform, that randomizes binary code at a per-page granularity level. AVRAND is a software-only solution that preprocesses a binary firmware image, divides it into chunks which can be randomized and adds metadata for proper control-flow reconstruction. The bootloader section is modified to include the randomization code in question and reflashes the randomized firmware image. The AVRAND approach does not meet our criteria since it imposes an average code size increase of 20%, requires sizable metadata storage in *EEPROM* and has the same device lifespan reduction drawback as MAVR stemming from its firmware reflashing approach. And while both the MAVR and AVRAND approaches, if ported from their original AVR-oriented designs, could avoid these drawbacks on a Von-Neumann style processor, the µArmor criteria require a hardware-agnostic approach suitable to both (modified) Harvard and Von Neumann processors.

There is a large body of work, of which ASLR is a part, on using **software diversification** [418] for probabilistic security purposes. With regards to using software diversity to complicate code-reuse attacks on embedded systems, prior works has proposed diversification of system calls [284] and APIs [441].

Most Linux-, Windows- and BSD-based embedded operating systems support some sort of **stack canary** scheme and if they don't it can usually be trivially ported. When it comes to RTOSes, however, stack canary support is minimal with *Zephyr* being the only RTOS in our sample set from Section 4.1 having canary support. Embedded-oriented alternatives for stack canaries have been proposed in the form of hardware-facilitated separate return stacks [72] and saved return address encryption [448] but the former does not meet our hardware agnosticism criteria while the latter does not allow for availability preservation.

There exists a significant body of work on **embedded random number generation**. Mobile operating systems tend to be Linux-, Windows- or BSD-based and as such inherit (a modified version of) their respective OS random number generators, though this is not without its problems as evidenced by attacks on the Android [326, 327], iOS [147] and Brillo [450] PRNGs. Unlike mobile devices, most embedded systems do not have the peripherals and user interaction from which to draw PRNG entropy. As such many embedded systems come equipped with on-board True Random Number Generators (TRNGs), the body of work on which is too large to summarize here. The problem with TRNGs is that a) they are not omnipresently available (due to cost, legacy hardware, etc.) and b) they tend to have low throughputs. As such entropy collection for secure random number generation is a significant problem in the embedded world [380, 407]. Prior work has tried to address this by exploring the possibilities offered by a variety of potential entropy sources such as sensor values [313, 396, 428], clock jitter and drift [377, 460, 461], ADC noise [18, 65, 305, 348], avalanche noise [304], wireless transmission bit errors [302] and SRAM startup values [96, 132, 298, 299, 319, 403].

Control-Flow Integrity (CFI) [408] is a technique that seeks to thwart control-flow hijacking attacks by checking whether program execution conforms to a valid control-flow graph (or a relaxed superset thereof) at runtime. The technique has seen much academic interest over the past years and variants of it have been rolled out in major compilers such as Microsoft's *Control-Flow Guard (CFG)* [142, 235, 272], Clang's CFI [473] and grsecurity GCC RAP [474]. In the embedded world, there have been CFI proposals targeted at mobile operating systems [368, 391], real-time operating systems [28, 338], Industrial Control Systems [376] and constrained embedded systems in general [392, 411]. Of these works `HCFI` [411], `HAFIX` [392] and `OCFMM` [338] are all hardware based and as such do not meet our mitigation criteria, while `μShield` [376], `MoCFI` [391] and `CFR` [368] are all designed for Linux- or BSD-based operating systems running on high-end embedded systems. The work in [28] proposes a hardware agnostic CFI implementation for bare-metal or RTOS-running constrained embedded systems but imposes 30% execution time and just under 10% code size overheads and thus does not meet our overhead criteria.

Execute-Only Memory (XoM), **Execute-no-Read (XnR)** and **No-Execute-After-Read (NEAR)** [367, 383, 401, 445, 446] are a collection of security techniques that prevent code from being read (or read memory from being subsequently execute in the case of `NEAR`) in order to prevent information leaks or IP theft and acts as a strong complementary protection to software diversification schemes. These techniques can be implemented purely in software or by using hardware support [89]. While prior work exists exploring these techniques on mobile operating systems such as Android [367], no work seems to exist exploring it for RTOSes and *deeply embedded systems*.

Firmware Integrity Verification seeks to protect against malicious firmware modifications resulting from vulnerabilities in the firmware update process, physical tampering attacks or vulnerabilities in the firmware itself. It does this by cryptographically verifying code and data integrity at a given moment. The body of work around firmware integrity verification is too big and diverse to summarize here, but verification techniques can be applied at different operational stages such as by means of secure (over-the-air) updating [329, 345, 406], secure boot [75, 149, 300, 370] and runtime verification [340, 343, 389, 463]. While firmware integrity verification does not seek to protect against memory corruption exploitation, some techniques might complicate exploitation by prohibiting code modification attacks or persistence payloads.

Ideally, memory corruption vulnerabilities would be prevented by introduction of omnipresent **memory safety** in embedded systems. While, as discussed in Section 2.4, unsafe languages dominate embedded systems development, safe languages like `Ada` and `Rust` are well-suited for embedded development purposes:

`Ada` is a strongly typed, imperative, object-oriented language derived from `Pascal`, `ALGOL 68` and other languages. It was designed in the late 1970s for the US Department of Defense for use in embedded and real-time systems with high safety requirements and has built-in support for design-by-contract and numerous compile-time and run-time safety checks. `Ada`, and its safety-critical oriented dialects such as `Ravenscar` and `SPARK`, is used in a wide range of sensitive systems including avionics, railway transportation, banking and military and aerospace applications [63, 362, 363]. Despite the availability and adoption of safe languages like `Ada` in critical embedded systems, two trends complicated the picture: a) certain low-level code (such as device drivers or software for bare metal systems) still tended to be written in unsafe languages even in `Ada`-dominant systems [363] and b) `Ada` has slowly been replaced by unsafe languages in many industry verticals. The latter is illustrated by the US DoD withdrawing `Ada` support during the 1990s and permitting the use of languages such as C(++), with the bulk of F-35 application code written in C++ and software for the F-16E/F even being ported from `Ada` to C++ [362]. There are many reasons for this shift but factors like a desire for increased commercial COTS adoption, performance, longevity and a lack of qualified programmers all play an important role.

`Rust` is a programming language that first appeared in 2010 and is aimed at creating safe and highly concurrent systems. It supports functional and imperative-procedural paradigms and is well-suited for low-level programming (eg. by providing memory safety without garbage collector [47]). While maturity and adoption in the embedded world are still in the early stages, there is promising work such as the *Rust Embedded* project [476], operating systems such as `Tock OS` [239, 290] and `intermezzOS` [117], firmware development [246], the ARM software stack `Zinc` [480] and safe and efficient parsers [46] all written either exclusively in `Rust` or with a minimal amount of C and assembly mixed in.

In addition, projects like `Safe TinyOS` [409] seek to provide memory safety by means of a modified toolchain producing code that enforces memory and type safety at runtime but requires code annotation by programmers thus introducing extra development effort and leaving room for error.

11 DISCUSSION, CONCLUSIONS & FUTURE WORK

11.1 DISCUSSION

The **research goal** of this work was to summarize the state-of-the-art in embedded operating system binary security and contribute to improve the security of embedded systems, in particular against memory corruption attacks. In order to achieve this goal we posed and answered four **research questions**:

RQ-1: *What would a minimum exploit mitigation baseline for embedded systems look like?*

In Chapter 3 we established a minimum exploit mitigation baseline based on the three mitigations most commonly integrated in modern general purpose OSes and toolchains: *Executable Space Protection (ESP)*, *Address Space Layout Randomization (ASLR)* and *Stack Canaries*.

These mitigations complement each other and are well adopted in the GP world and as such provide a suitable absolute minimum baseline for embedded systems. We outlined their security requirements based on prior work and subsequently mapped out the hardware and software dependencies of these mitigations in order to facilitate our quantitative and qualitative analyses.

RQ-2: *What is the current state-of-the-art in embedded operating system exploit mitigations in terms of adoption, dependency support and implementation quality?*

In order to get an idea of the state of embedded binary security, we performed the first quantitative analysis of embedded mitigation and dependency support in Chapter 4. This analysis used a selection of 41 popular embedded operating systems and 78 popular embedded processor core families and shows that while mobile operating systems have support for all mitigations in our baseline, the rest does not. In particular, operating systems not based on Linux, Windows or BSD, tend to lack support for most mitigations and often don't support the required hardware- and software dependencies either. Especially so-called *low-end* operating systems offer no support for crucial dependencies such as *virtual memory* or a *secure random number generator*. When it comes to embedded processors, many lack crucial dependencies like *MMUs* or *hardware ESP support*. While the latter is increasingly common on modern 32-bit processors the former is not, especially for various types of microcontrollers.

The history of existing mitigations in the general purpose world and the level of scrutiny they have been exposed to over the years means that porting them to Linux-, Windows- and BSD-based embedded operating systems is usually easier and results in more robust mitigations than for operating systems which do not share this lineage. For those OSes, mitigations have to be implemented from scratch to fit within architectures not designed to accommodate them. In Chapter 5 we performed the first qualitative analysis of the exploit mitigations and OS CSPRNGs of three non Linux-, Windows- or BSD-based embedded operating systems: `QNX`, `RedactedOS` and `Zephyr`. We discovered that all of them suffer from various flaws, ranging from insecure default settings, insecure random number generation and information leaks to incomplete implementations. We responsibly disclosed these issues to the vendors and collaborated in drafting fixes, resulting in `QNX` releasing patches for 6.6 and redesigning parts of `QNX` 7.0 to incorporate corresponding improvements.

RQ-3: *What are the gap areas and open problems within the current state-of-the-art and what are the challenges underlying them?*

In Chapter 6 we explained the observed gaps in mitigation support and quality among embedded operating systems by identifying a range of challenges. First of all, incentive issues are a major challenge for the embedded world as exemplified by the heavy focus on time-to-market and novel features as well as the fact that most vendors' security concerns relate to the company rather than the product user. Secondly, the embedded development landscape is very diverse and fragmented which complicates mitigation adoption. On top of that, cost cutting practices result in cheap hardware with limited resources and often make it hard to justify the cost of security improvements. Finally, safety, reliability, and real-time requirements and OS and hardware limitations mean many of the features that mitigations depend on (such as MMUs, virtual memory or OS CSPRNGs) are not be widely adopted. Based on those challenges and the results of *RQ-2*, we identified two pressing open problems: a) *mitigation designs for deeply embedded systems* and b) *deeply embedded OS CSPRNG design*.

RQ-4: *Given the clearest gap area identified, what would an effective solution look like and what criteria should it meet?*

In Section 6.5.1, we outlined the criteria mitigation designs for deeply embedded systems should meet: 1) *Limited Resource Pressure*, 2) *Hardware Agnostic*, 3) *Availability Preservation*, 4) *Real-Time Friendly*, and 5) *Easy (RT)OS Integration*.

In Section 6.5.2, we outlined the criteria OS CSPRNG designs for deeply embedded systems should meet: 1) *Lightweight Cryptography*, 2) *Entropy Gathering Limitations*, and 3) *Non-Domain Specific Entropy Sources*.

In Chapter 7 we designed µ`Armor` to meet these criteria: the first exploit mitigation baseline for constrained embedded systems running a low-end operating system lacking virtual memory support and having either a Harvard-style processor or a Von-Neumann one with hardware ESP support. µ`Armor` consists of µ`ESP` , µ`Scramble` , µ`SSP` and µ`RNG` being an ESP component, a compile- and update-time diversification component providing address randomization, a stack canary component and an OS CSPRNG component respectively.

In Chapter 8 we showed how we implemented µ`Armor` for the *Zephyr* RTOS on the `TI LM3S6965` microcontroller. µ`ESP` was implemented using the ARM Cortex-M3 MPU, µ`Scramble` was implemented as a collection of *LLVM* compiler passes, µ`SSP` was implemented by modifying the pre-existing *Zephyr* stack canary scheme to support a modular violation handler and µ`RNG` was implemented as a modification of a `Keccak`-based PRNG using entropy sources omnipresent on microcontrollers such as SRAM startup values and clock jitter/drift.

Finally, in Chapter 9, we performed an overhead evaluation of µ`Armor` and found it to impose overheads of at most 5% for code size and below 1% for data size, memory usage, and runtime in all cases. We also performed a security evaluation in the form of a theoretical analysis for all components as well as an additional coverage analysis for µ`Scramble` , all of which found µ`Armor` to hold up to our attacker model.

11.2 CONCLUSIONS

Memory corruption issues are one of the most common vulnerabilities affecting embedded devices and are further compounded by embedded patching issues which cause prolonged (sometimes indefinite) vulnerability exposure windows. This is worrying given the critical nature of many embedded systems and their projected growth and increasing interconnectedness with the rise of the IoT. Ideally, the issue of memory corruption is addressed by getting to the root of the problem and using safe languages. But embedded development practices continue to be dominated by unsafe languages like C(++). As such, there is a clear need for short-term solutions which at least reduces the problem's scale and impact. We believe exploit mitigations, while not a *'silver bullet'*, to be such a solution and consider a minimum mitigation baseline to consist of *ESP*, *ASLR* and *Stack Canaries*. This opinion is confirmed by the track record of exploit mitigations in the general-purpose world.

While Linux-, Windows- and BSD-based embedded operating systems (*mobile operating systems* in particular) tend to support this minimum baseline, there is significantly less support for it in OSs used in embedded systems. And among the low-end operating systems targeted at deeply embedded systems, there is basically no support for any mitigations nor for most of their software dependencies. On top of that, many embedded processors lack the hardware features required to support this baseline. When it comes to the quality of mitigation implementations on non Linux-, Windows- and BSD-based embedded OSes, we can see that this too tends to be worse than it is in general-purpose operating systems. As such, we can conclude that embedded binary security lags behind the general purpose world significantly.

However, porting existing exploit mitigation designs from the GP world or designing completely new ones will not be trivial due to a myriad of challenges and limitations inherent to the embedded world. One particular 'open problem' area are deeply embedded systems, for which there are no existing exploit mitigation designs. We outlined the criteria for such designs and proposed `µArmor` as a first step towards a viable exploit mitigation baseline for deeply embedded systems.

11.3 FUTURE WORK

We see two main trajectories for future work on embedded binary security: *long-term* and *short-term* solutions. The former trajectory aims to develop robust techniques tackling the problem at the root and requires changes along the whole development chain. Examples are increased adoption and expansion of the capabilities of embedded-oriented safe languages and secure and scalable patching solutions.

The short-term trajectory should aim to develop solutions which reduce the impact of embedded memory corruption vulnerabilities and which can be rapidly adopted. Examples are scalable vulnerability discovery and exploit mitigation design for embedded systems. The latter includes addressing the limitations of `µArmor` such as extending it to support Von Neumann processors without hardware ESP, exploring initial entropy alternatives to SRAM Startup Values for the subset of chips without suitable entropy and providing more robust address randomization (eg. higher relocation frequency, covering both code and data memory, etc.). It also includes the exploration of more advanced mitigations for embedded systems in order to continue raising the bar and close the binary security gap between general purpose and embedded systems.

Part V

APPENDIX

A SUPPLEMENTARY DATA

Board	TRNG	Architecture
Arduino/Genuino 101 (Application)	×	x86
Galileo Gen1/Gen2	×	x86
MinnowBoard Max	✓	x86
Quark D2000 Development Board	×	x86
tinyTILE	×	x86
96Boards Carbon	✓	ARM
96Boards Carbon nRF51	×	ARM
96Boards Nitrogen	×	ARM
Arduino Due	✓	ARM
CC3200 LaunchXL	×	ARM
CC3220SF LaunchXL	✓	ARM
ST Disco L475 IOT01	✓	ARM
NXP FRDM-K64F	✓	ARM
NXP FRDM-KL25Z	×	ARM
NXP FRDM-KW41Z	✓	ARM
Hexiwear	✓	ARM
Hexiwear KW40Z	✓	ARM
ARM V2M MPS2	×	ARM
nRF51-PCA10028	×	ARM
nRF52840-PCA10056	✓	ARM
nRF52-PCA10040	×	ARM
Redbear Labs Nano v2	×	ARM
ST Nucleo F334R8	×	ARM
ST Nucleo F401RE	✓	ARM
ST Nucleo F411RE	✓	ARM
ST Nucleo F412ZG	✓	ARM
ST Nucleo F413ZH	✓	ARM
ST Nucleo L432KC	✓	ARM
ST Nucleo L476RG	✓	ARM
OLIMEX-STM32-E407	✓	ARM
OLIMEXINO-STM32	×	ARM
SAM4S Xplained	×	ARM
SAM E70 Xplained	✓	ARM
STM3210C-EVAL	×	ARM
STM32373C-EVAL	×	ARM
ST STM32F469I Discovery	✓	ARM
ST STM32F4DISCOVERY	✓	ARM
ST STM32L496G Discovery	✓	ARM
ARM V2M Beetle	✓	ARM
Arduino/Genuino 101 (Sensor Subsystem)	×	ARC
DesignWare ARC EM Starter Kit	×	ARC
Altera MAX10	×	NIOS II

Table 27: TRNG support among Zephyr 1.8 supported boards

Board	Speed	Flash	(S)RAM	Storage
General				
Arduino Uno	16 MHz	32 KB	2 KB	1 KB EEPROM
BLE NANO v1.5	16 MHZ	256 KB	32 KB	×
Arduino 101[1]	32 MHz	196 KB	24 KB	×
Arduino 101[2]	32 MHz	×	×	×
RPi Zero W	1 GHz	×	512 MB	MicroSDHC
RPi Model B 3	1.2 GHz	×	1 GB	MicroSDHC
Industrial				
Controllino Maxi	16 MHz	64-256 KB	8 KB	×
Arduino Industrial[3]	16 MHz	32 KB	2.5 KB	1 KB EEPROM
Arduino Industrial[4]	400 MHz	16 MB	64 MB	×
UniPi Neuron S103	1.2 GHz	×	1 GB	MicroSDHC
Wearables				
Arduino Gemma	8 MHz	8 KB	512 B	512 B EEPROM
Hexiwear[5]	48 MHz	160 KB	20 KB	×
Hexiwear[6]	120 MHz	1 MB (+8 MB)	256 KB	×

Table 28: Popular Embedded Development Board Resources
[1] *Intel Quark SE Core*, [2] *ARC EM Core*, [3] *ATmega32U4 Core*, [4] *AR9331 Core*, [5] *Cortex-M0+ Core*, [6] *Cortex-M4 Core*

Name	**Desc.**	**SLOC**[1]
Application		
`lift`	Lift controller	361
`powerwindow`	Distributed power window control	2533
Kernel		
`binarysearch`	Binary search of 15 integers	47
`bitcount`	Couting number of bits in an integer array	164
`bitonic`	Bitonic sorting network	52
`bsort`	Bubblesort program	32
`complex_updates`	Multiply-add with complex vectors	18
`countnegative`	Counts signes in a matrix	35
`fac`	Factorial function	21
`fft`	1024-point FFT, 13 bits per twiddle	78
`filterbank`	Filter bank for multirate signals	75
`fir2dim`	2-dimensional FIR filter convolution	75
`iir`	Biquad IIR 4 sections filter	27
`insertsort`	Insertion sort	35
`jfdctint`	Discrete-cosine transformation on a 8x8 pixel block	123
`lms`	LMS adaptive signal enhancement	51
`ludcmp`	LU decomposition	68
`matrix1`	Generic matrix multiplication	28
`md5`	Message digest algorithm	344
`minver`	Floating point matrix inversion	141
`pm`	Pattern match kernel	484
`prime`	Prime number test	41
`quicksort`	Quick sort of strings and vectors	992
`recursion`	Artificial recursive code	18
`sha`	NIST secure hash algorithm	382
`st`	Statistics calculations	90

Table 29: Benchmarks & Applications selected from TACLeBench Suite [357]
[1] *Source Lines of Code (SLOC)*

Name	Desc.	SLOC[1]
Sequential		
adpcm_dec	ADPCM decoder	293
adpcm_enc	ADPCM encoder	316
ammunition	C compiler arithmetic stress test	2431
anagram	Word anagram computation	2710
audiobeam	Audio beam former	833
cjpeg_transupp	JPEG image transcoding routines	608
cjpeg_wrbmp	JPEG image bitmap writing code	892
dijkstra	All pairs shortest path	117
epic	Efficient pyramid image coder	451
fmref	Software FM radio with equalizer	680
gsm_dec	GSM provisional standard decoder	543
h264_dec	H.264 block decoding functions	460
huff_dec	Huffman decoding with a file source to decompress	183
huff_enc	Huffman encoding with a file source to compress	325
mpeg2	MPEG2 motion estimation	1297
ndes	Complex embedded code	260
petrinet	Petri net simulation	500
rijndael_dec	Rijndael AES decryption	820
rijndael_enc	Rijndael AES encryption	734
statemate	Statechart simulation of a car window lift control	1038
Test		
cover	Artificial code with lots of different control flow paths	620
duff	Duff's device	35
test3	Artificial WCET analysis stress test	4235

Table 30: Benchmarks & Applications selected from TACLeBench Suite [357]
[1] *Source Lines of Code (SLOC)*

Listing 2: QNX vmm_mmap Routine

```
int vmm_mmap(PROCESS *prp, uintptr_t vaddr_requested, size_t
    size_requested,
        int prot, int flags, OBJECT *obp, uint64_t boff, unsigned
             alignval,
        unsigned preload, int fd, void **vaddrp, size_t *sizep,
            part_id_t mpart_id)
{
    ...

    create_flags = flags;

    ...

    if ( prp->flags & _NTO_PF_ASLR )
        create_flags |= MAP_SPARE1;

    r = map_create(..., create_flags);
}
```

Listing 3: QNX map_create Routine

```
int map_create(struct map_set *ms, struct map_set *repl, struct
    mm_map_head *mh,
        uintptr_t va, uintptr_t size, uintptr_t mask, unsigned
            flags)
{
    ...

    if(flags & (MAP_FIXED|IMAP_GLOBAL)) {
        ...
    } else {
        repl->first = NULL;

        va = map_find_va(mh, va, size, mask, flags);
        if(va == VA_INVALID) {
            r = ENOMEM;
            goto fail1;
        }
    }

    ...
}
```

Listing 4: QNX map_find_va Routine

```c
uintptr_t map_find_va(struct mm_map_head *mh, uintptr_t va,
    uintptr_t size,
        uintptr_t mask, unsigned flags)
{

    sz_val = size - 1;

    ...

    if ( flags & MAP_SPARE1 )
    {
        uint64_t clk_val = ClockCycles();
        unsigned int rnd_val = ((_DWORD)clk_val << 12) & 0xFFFFFF
            ;

        if ( flags & MAP_BELOW )
        {
            start_distance = start - best_start;
            if ( start != best_start )
            {
                if ( rnd_val > start_distance )
                    rnd_val %= start_distance;
                start -= rnd_val;
            }
        }
        else
        {
            end_distance = best_end - sz_val - start;
            if ( best_end - sz_val != start )
            {
                if ( rnd_val > end_distance )
                    rnd_val %= end_distance;
                start += rnd_val;
            }
        }
    }
```

Listing 5: QNX stack_randomize Routine

```
uintptr_t stack_randomize(const THREAD *const thp, uintptr_t
    new_sp)
{
    uintptr_t rnd_sp;
    size_t stack_size;
    unsigned int size_mask;

    rnd_sp = new_sp;

    if ( thp->process->flags & _NTO_PF_ASLR )
    {
        stack_size = thp->un.lcl.stacksize >> 4;
        if ( stack_size )
        {
            size_mask = 0x7FF;
            if ( stack_size <= 0x7FE )
                do { size_mask >>= 1; } while ( size_mask >
                    stack_size );

            rnd_sp = (new_sp - ((ClockCycles() << 4) & size_mask)
                ) & 0xFFFFFFF0;
        }
    }
    return rnd_sp;
}
```

Listing 6: QNX Stack Canary Failure Handler (User-Space)

```
void __stack_chk_fail(void)
{
        if ((fd = open("/dev/tty", 1)) != -1)
                write(fd, "*** stack smashing detected ***");
        raise(SIGABRT);
}
```

Listing 7: QNX Stack Canary Generation Handler

```c
void _init_cookies(void)
{
        void* stackval;

        ts0 = (ClockCycles() & 0xffffffff);
        canary0 = (ts0 ^ (((&stackval) ^ (_init_cookies)) >> 8));
        _stack_chk_guard = canary0;

        ts1 = (ClockCycles() & 0xffffffff);
        canary1 = (((&stackval) ^ canary0) >> 8);
        _atexit_list_cookie = (canary1 ^ ts1);

        ts2 = (ClockCycles() & 0xffffffff);
        _atqexit_list_cookie = (canary1 ^ ts2);

        _stack_chk_guard &= 0xff00ffff;
}
```

Listing 8: QNX Stack Canary Failure Handler (Kernel-Space)

```c
void __stack_chk_fail(void)
{
        kprintf("*** stack smashing detected in procnto ***");
        __asm{ int 0x22 };
}
```

Listing 9: QNX Yarrow HPC Entropy Collection Snippet

```c
if( Yarrow )
{
        yarrow_output( Yarrow, (uint8_t *)&rdata, sizeof( rdata )
             );
        timeout = ( rdata & 0x3FF ) + 10;
}

delay( timeout );
clk = ClockCycles();
clk = clk ^ rdata;

if( Yarrow )
        yarrow_input( Yarrow, (uint8_t *)&clk, sizeof( clk ),
            pool_id, 8 );
```

Listing 10: QNX 6.6.0 yarrow_do_sha1 function

```
void yarrow_do_sha1( yarrow_t *p, yarrow_gen_ctx_t *ctx )
{
        SHA1Init(&sha);

        IncGaloisCounter5X32(p->pool.state);
        sha.state[0] ^= p->pool.state[4];
        sha.state[1] ^= p->pool.state[3];
        sha.state[2] ^= p->pool.state[2];
        sha.state[3] ^= p->pool.state[1];
        sha.state[4] ^= p->pool.state[0];

        SHA1Update(&sha, ctx->iv, 20);
        SHA1Update(&sha, ctx->out, 20);
        SHA1Final(ctx->out, &sha);
}
```

Listing 11: QNX 6.6.0 yarrow_make_new_state function

```
void yarrow_make_new_state(yarrow_t *p, yarrow_gen_ctx_t *ctx,
    uint8_t *state )
{
  for(i = 0; i < 20; i++)
    ctx->iv[i] ^= state

  SHA1Init(&sha);

  IncGaloisCounter5X32(p->pool.state);
  sha.state[0] ^= p->pool.state[4];
  sha.state[1] ^= p->pool.state[3];
  sha.state[2] ^= p->pool.state[2];
  sha.state[3] ^= p->pool.state[1];
  sha.state[4] ^= p->pool.state[0];

  SHA1Update(&sha, ctx->iv, 20);
  SHA1Final(ctx->out, &sha);
}
```

Listing 12: Zephyr Stack Canary Generation Handler

```
FUNC_NORETURN void _Cstart(void)
{
    ...

    /* perform basic hardware initialization */
    _sys_device_do_config_level(_SYS_INIT_LEVEL_PRE_KERNEL_1);
    _sys_device_do_config_level(_SYS_INIT_LEVEL_PRE_KERNEL_2);

    /* initialize stack canaries */
#ifdef CONFIG_STACK_CANARIES
    __stack_chk_guard = (void *)sys_rand32_get();
#endif
    ...
}
```

Listing 13: Zephyr Timer Random Number Generator

```
static atomic_val_t _rand32_counter;

#define _RAND32_INC 1000000013

uint32_t sys_rand32_get(void)
{
    return k_cycle_get_32() + atomic_add(&_rand32_counter,
        _RAND32_INC);
}
```

BIBLIOGRAPHY

[1] Aditya K Sood (oknock). *Digging Inside the VxWorks OS and Firmware The Holistic Security*. 2011. URL: http://www.secniche.org/vxworks/vxworks_os_holistic_security_adityaks.pdf.

[2] ARM. *ARM Cortex-M3 Memory Protection Unit (MPU)*. 2017. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0179b/CHDFDFIG.html.

[3] ARM. *ARM TrustZone*. 2017. URL: https://www.arm.com/products/security-on-arm/trustzone.

[4] ARM. *PUSH and POP*. 2017. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Babefbce.html.

[5] Mustafa Al-Bassam. *Equation Group Firewall Operations Catalogue*. 2016. URL: https://musalbas.com/2016/08/16/equation-group-firewall-operations-catalogue.html.

[6] Julia Alexander. *The Prosthetic DEKA Arm Is Hackable and a Legal Mess*. 2014. URL: https://motherboard.vice.com/en_us/article/the-deka-arm-is-hackable-and-that-might-open-up-a-legal-can-of-worms.

[7] Cloud Security Alliance. *Future-proofing the Connected World: 13 Steps to Developing Secure IoT Products*. 2016. URL: https://downloads.cloudsecurityalliance.org/assets/research/internet-of-things/future-proofing-the-connected-world.pdf.

[8] Ross Anderson. "Protecting embedded systems the next ten years." In: *CHES* (2001).

[9] Alexander Antukh. *Dissecting Blackberry 10 – An initial analysis*. 2013. URL: https://www.sec-consult.com/fxdata/seccons/prod/downloads/sec_consult_vulnerability_lab_blackberry_z10_initial_analysis_v10.pdf.

[10] Charles Arthur. *SkyGrabber: the $26 software used by insurgents to hack into US drones*. 2009. URL: https://www.theguardian.com/technology/2009/dec/17/skygrabber-software-drones-hacked.

[11] Atmel. *AT11787: Safe and Secure Firmware Upgrade via Ethernet*. 2015. URL: http://www.atmel.com/Images/Atmel-42492-Safe-and-Secure-Firmware-Upgrade-via-Ethernet_ApplicationNote_AT11787.pdf.

[12] Christian Backer. *Digital Forensics on Small Scale Digital Devices*. 2009. URL: https://www.emsec.rub.de/media/crypto/attachments/files/2011/03/baecker_digital_forensics.pdf.

[13] Michael Barr. *Trends in Embedded Software Design*. 2012. URL: http://embeddedgurus.com/barr-code/2012/04/trends-in-embedded-software-design/.

[14] Michael Barr. *An Update on Toyota and Unintended Acceleration*. 2013. URL: http://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/.

[15] Michael Barr. *A Look Back at the Audi 5000 and Unintended Acceleration*. 2014. URL: http://embeddedgurus.com/barr-code/2014/03/a-look-back-at-the-audi-5000-and-unintended-acceleration/.

[16] Michael Barr. *Lethal Software Defects: Patriot Missile Failure*. 2014. URL: http://embeddedgurus.com/barr-code/2014/03/lethal-software-defects-patriot-missile-failure/.

[17] Michael Barr. *Government-Sponsored Hacking of Embedded Systems*. 2015. URL: http://embeddedgurus.com/barr-code/2015/03/government-sponsored-hacking-of-embedded-systems/.

[18] Johannes Bauer. *On Inexpensive Methods for Improving Security of Embedded Systems*. 2016. URL: https://www.johannes-bauer.com/personal/publications/2016-11-Bauer-PhDThesis.pdf.

[19] Hanan Be'er. *Metaphor: A (real) reallife Stagefright exploit*. 2016. URL: https://www.exploit-db.com/docs/39527.pdf.

[20] Konstantin Belousov. *FreeBSD ABI: Shared Page*. 2012. URL: http://kib.kiev.ua/kib/kievbsd-sharedpage.pdf.

[21] Dillon Beresford. *Exploiting Siemens Simatic S7 PLCs*. 2011. URL: https://media.blackhat.com/bh-us-11/Beresford/BH_US11_Beresford_S7_PLCs_WP.pdf.

[22] Patrick Biernat. *Address Space Layout Random Randomization*. 2015. URL: http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/15/09_lecture.pdf.

[23] BlackBerry. *Using compiler and linker defenses (BlackBerry Native SDK for PlayBook OS)*. URL: http://developer.blackberry.com/playbook/native/reference/com.qnx.doc.native_sdk.security/topic/using_compiler_linker_defenses.html.

[24] BlackBerry. *QNX*. 2017. URL: http://www.qnx.com/content/qnx/en.html.

[25] Quarkslab Blog. *Clang Hardening Cheat Sheet*. 2016. URL: http://blog.quarkslab.com/clang-hardening-cheat-sheet.html.

[26] Gerrit De Vynck Bloomberg. *CIA Listed BlackBerry's Car Software as Possible Target*. 2017. URL: https://www.bloomberg.com/news/articles/2017-03-08/cia-listed-blackberry-s-car-software-as-possible-target-in-leak.

[27] Cody Brocious. *My Arduino can beat up your hotel room lock*. 2012. URL: http://demoseen.com/bhtalk2.pdf.

[28] Nicholas Brown. *Control-flow Integrity for Real-time Embedded Systems*. 2017. URL: https://web.wpi.edu/Pubs/ETD/Available/etd-042717-120631/unrestricted/nfbrown-thesis-final.pdf.

[29] Robert G. Brown. *Dieharder: A Random Number Test Suite*. URL: https://www.phy.duke.edu/\~rgb/General/dieharder.php.

[30] Tim Brown. *QNX Advisories*. URL: https://packetstormsecurity.com/files/author/4309/.

[31] Jared Candelaria. *Modern Userland Exploitation: From gets() to pwned*. 2013. URL: http://security.cs.rpi.edu/~candej2/user/userland_exploitation.pdf.

[32] Evan Baldwin Carr. "Unmanned Aerial Vehicles: Examining the Safety, Security, Privacy and Regulatory Issues of Integration into U.S. Airspace." In: (). URL: http://www.ncpa.org/pdfs/sp-Drones-long-paper.pdf.

[33] Silvio Cesare. *CVE-2004-1453*. 2004. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1453.

[34] Silvio Cesare. *Breaking the Security of Physical Devices*. 2014. URL: https://regmedia.co.uk/2014/08/06/dfgvhbhjkui867ujk5ytghj.pdf.

[35] Liang Chen. *Webkit Everywhere: Secure or Not?* 2014. URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Chen-WebKit-Everywhere-Secure-Or-Not-WP.pdf.

[36] Liat Clark. *Watch hackers take over a Tesla's brakes from 12 miles away*. 2016. URL: http://www.wired.co.uk/article/tesla-remotely-hacked-by-chinese-collective-keen.

[37] Peter Clarke. *MCU market turns to 32-bits and ARM*. 2013. URL: http://www.eetimes.com/document.asp?doc_id=1280803.

[38] Frederick Cohen. "Operating system protection through program evolution." In: *Computers and Security* (1993).

[39] The Institute for Communication Technologies and Embedded Systems (ICE). *DSPstone: DSP Compiler and Processor Evaluation*. 2017. URL: https://www.ice.rwth-aachen.de/research/tools-projects/entry/detail/dspstone/.

[40] MediaBench Consortium. *MediaBench*. 2017. URL: http://mathstat.slu.edu/~fritts/mediabench/.

[41] SEC Consult. *House of Keys: Industry-Wide HTTPS Certificate and SSH Key Reuse Endangers Millions of Devices Worldwide*. 2015. URL: `http://blog.sec-consult.com/2015/11/house-of-keys-industry-wide-https.html`.

[42] Microsoft Corporation. *The BlueHat Prize*. 2012. URL: `https://www.microsoft.com/security/bluehatprize/`.

[43] Aldo Cortesi. *binvis.io: visual analysis of binary files*. URL: `http://binvis.io`.

[44] Andrei Costin. *Poor Man's Panopticon Mass CCTV Surveillance for the masses*. 2013. URL: `http://andreicostin.com/papers/poc2013_andrei.slides.pdf`.

[45] Counterpane. *Yarrow 0.8.71*. URL: `https://www.schneier.com/code/Yarrow0.8.71.zip`.

[46] Geoffoy Couprie. "Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust." In: *IEEE CS Security and Privacy Workshops* (2015).

[47] Yan Cui. *Rust - memory safety without garbage collector*. 2015. URL: `http://theburningmonk.com/2015/05/rust-memory-safety-without-gc/`.

[48] Debian. *Hardening*. 2017. URL: `https://wiki.debian.org/Hardening`.

[49] Dell. *2015 Dell Security Annual Threat Report*. 2015.

[50] CVE Details. *QNX CVEs*. URL: `https://www.cvedetails.com/vulnerability-list/vendor_id-436/QNX.html`.

[51] Will Dormann. *Feeling Insecure? Blame Your Parent!* 2014. URL: `https://insights.sei.cmu.edu/cert/2014/02/feeling-insecure-blame-your-parent.html`.

[52] Mark Dowd. *The (Memory Corruption) Safety Dance*. 2017. URL: `https://www.youtube.com/watch?v=r2nVZ9BOAKo`.

[53] Thomas Dullien. *Exploitation and state machines: Programming the "weird machine", revisited*. 2011.

[54] Thomas Dullien. *Understanding the fundamentals of attacks: What is happening when someone writes an exploit?* 2016.

[55] ENISA. *Hardware Threat Landscape and Good Practice Guide*. 2017. URL: `https://www.enisa.europa.eu/publications/hardware-threat-landscape`.

[56] Infineon – NXP – STMicroelectronics – ENISA. *Common Position On Cybersecurity*. 2016. URL: `https://www.enisa.europa.eu/publications/enisa-position-papers-and-opinions/infineon-nxp-st-enisa-position-on-cybersecurity`.

[57] Jake Edge. *proc: avoid information leaks to non-privileged processes*. 2009. URL: `https://patchwork.kernel.org/patch/21766/`.

[58] Stefan Esser. *iOS 678 Security: A Study in Fail*. 2015. URL: https://www.slideshare.net/i0n1c/syscan-2015-esserios678securityastudyinfail.

[59] Stephen Evanczuk. *MCU popularity perceptions*. 2013. URL: http://www.edn.com/electronics-blogs/systems-interface/4420230/MCU-popularity-perceptions.

[60] Chris Evans. *Some random observations on Linux ASLR*. 2012. URL: https://scarybeastsecurity.blogspot.nl/2012/03/some-random-observations-on-linux-aslr.html.

[61] Chris Evans. *Are we doing memory corruption mitigations wrong?* 2017. URL: https://scarybeastsecurity.blogspot.nl/2017/05/are-we-doing-memory-corruption.html.

[62] Eric Evenchick. *An Introduction to the CANard Toolkit*. 2015. URL: https://www.blackhat.com/docs/asia-15/materials/asia-15-Evenchick-Hopping-On-The-Can-Bus-wp.pdf.

[63] Michael B. Feldman. *Who's Using Ada? Real-World Projects Powered by the Ada Programming Language*. 2014. URL: http://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html.

[64] Justin Fier. *The False Binary of IoT and Traditional Cyber Security*. 2017. URL: http://www.securityweek.com/false-binary-iot-and-traditional-cyber-security.

[65] Patrik Fimml. *HOWTO: A Simple Random Number Generator for the ATmega1280 Microcontroller*. 2013. URL: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu_2014/misc/task1-specific-stuff/rand_howto.pdf.

[66] FireEye. *Overload: Critical Lessons from 15 Years of ICS Vulnerabilities*. 2016.

[67] Yannick Formaggio. *Attacking VxWorks: from Stone Age to Interstellar*. 2015. URL: https://www.slideshare.net/44Con/44con-london-attacking-vxworks-from-stone-age-to-interstellar.

[68] Julio Cesar Fort. *QNX Advisories*. URL: https://packetstormsecurity.com/files/author/3551/.

[69] The Linux Foundation. *The Linux Foundation*. URL: https://www.linuxfoundation.org/.

[70] Thomas Fox-Brewster. *Here's How The CIA Allegedly Hacked Samsung Smart TVs – And How To Protect Yourself*. 2017. URL: https://www.forbes.com/sites/thomasbrewster/2017/03/07/cia-wikileaks-samsung-smart-tv-hack-security/.

[71] Thomas Fox-Brewster. *Medical Devices Hit By Ransomware For The First Time In US Hospitals*. 2017. URL: https://www.forbes.com/sites/thomasbrewster/2017/05/17/wannacry-ransomware-hit-real-medical-devices/.

[72] Aurelien Francillon. *Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks*. 2010. URL: https://tel.archives-ouvertes.fr/tel-00540371/PDF/these.pdf.

[73] Ingar Fredriksen. *Choosing a MCU for your next design; 8 bit or 32 bit?* 2014. URL: http://www.atmel.com/images/45107a-choosing-a-mcu-fredriksen_article_103114.pdf.

[74] Inc. Freescale Semiconductor. *K64 Sub-Family Reference Manual*. 2014. URL: http://www.nxp.com/docs/en/reference-manual/K66P144M180SF5RMV2.pdf.

[75] NXP Freescale. *Secure Boot: For QorIQ Communications Processors*. 2013. URL: http://www.nxp.com/assets/documents/data/en/white-papers/QORIQSECBOOTWP.pdf.

[76] Hagen Fritsch. *Buffer overflows on linux-x86-64*. 2009. URL: http://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Buffer-Overflows-Linux-whitepaper.pdf.

[77] Hagen Fritsch. *Stack Smashing as of Today*. 2009. URL: http://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf.

[78] Kaspersky GReAT. *Equation Group: Questions and Answers*. 2015. URL: https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf.

[79] Peter Geissler. "In Stickers We Trust: Breaking Naive ESSID/WPA2 Key Generation Algorithms." In: *HITB Amsterdam* (2016). URL: https://conference.hitb.org/hitbsecconf2016ams/materials/D2T2%20-%20Peter%20blasty%20Geissler%20-%20Breaking%20Naive%20ESSID%20WPA2%20Key%20Generation%20Algorithms.pdf.

[80] Chris Gerlinsky. *How Do I Crack Satellite and Cable Pay TV?* 2016. URL: https://media.ccc.de/v/33c3-8127-how_do_i_crack_satellite_and_cable_pay_tv.

[81] Hector Marco Gisbert. "Cyber-security protection techniques to mitigate memory errors exploitation." In: (2015). URL: https://riunet.upv.es/bitstream/handle/10251/57806/Marco%20-%20Cyber-security%20protection%20techniques%20to%20mitigate%20memory%20errors%20exploitation.pdf?sequence=1&isAllowed=y.

[82] Dan Goodin. *Insulin pump hack delivers fatal dosage over the air*. 2011. URL: https://www.theregister.co.uk/2011/10/27/fatal_insulin_pump_attack/.

[83] Dan Goodin. *Intruders hack industrial heating system using backdoor posted online*. 2012. URL: https://arstechnica.com/security/2012/12/intruders-hack-industrial-control-system-using-backdoor-exploit/.

[84] Dan Goodin. *Rise of "forever day" bugs in industrial systems threatens critical infrastructure*. 2012. URL: https://arstechnica.com/business/2012/04/rise-of-ics-forever-day-vulnerabiliities-threaten-critical-infrastructure/.

[85] Travis Goodspeed. *Nifty Tricks and Sage Advice for Shellcode on Embedded Systems*. 2013. URL: https://conference.hitb.org/hitbsecconf2013ams/materials/D1T1%20-%20Travis%20Goodspeed%20-%20Nifty%20Tricks%20and%20Sage%20Advice%20for%20Shellcode%20on%20Embedded%20Systems.pdf.

[86] Travis Goodspeed. *A Keynote in Praise of Junk Hacking*. 2016. URL: https://www.youtube.com/watch?v=ZmZ_tvbhJ0I.

[87] Adrian Graham. *Communications, Radar and Electronic Warfare*. Wiley, 2010.

[88] Barr Group. *Embedded Systems Safety & Security Survey 2017*. 2017. URL: https://barrgroup.com/Embedded-Systems/Surveys/2017-embedded-systems-safety-security-survey.

[89] ARM Compiler Software Development Guide. *2.21 Execute-only memory*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471m/chr1368698326509.html.

[90] ARM Cortex-M3 Devices Generic User Guide. *System control block*. 2017. URL: https://developer.arm.com/docs/dui0552/latest/4-cortex-m3-peripherals/43-system-control-block.

[91] Ben Hawkes. *What makes software exploitation hard?* 2016. URL: https://www.usenix.org/sites/default/files/conference/protected-files/enigma_slides_hawkes.pdf.

[92] Craig Heffner. *Exploiting Surveillance Cameras Like a Hollywood Hacker*. 2013. URL: https://media.blackhat.com/us-13/US-13-Heffner-Exploiting-Network-Surveillance-Cameras-Like-A-Hollywood-Hacker-WP.pdf.

[93] Craig Heffner. *Re-enabling JTAG and Debugging the WRT120N*. 2014. URL: http://www.devttys0.com/2014/02/re-enabling-jtag-and-debugging-the-wrt120n/.

[94] Alejandro Hernandez. *A Short Tale About executable_stack in elf_read_implies_exec() in the Linux Kernel*. URL: http://blog.ioactive.com/2013/11/a-short-tale-about-executablestack-in.html.

[95] Alejandro Hernández. *A Short Tale About executable_stack in elf_read_implies_exec() in the Linux Kernel*. 2013. URL: http://blog.ioactive.com/2013/11/a-short-tale-about-executablestack-in.html.

[96] Anthony Van Herrewege. *Lightweight PUF-based Key and Random Number Generation*. 2015. URL: https://www.esat.kuleuven.be/cosic/publications/thesis-254.pdf.

[97] U.S. Department of Homeland Security. *Common Cybersecurity Vulnerabilities in Industrial Control Systems*. 2011. URL: https://ics-cert.us-cert.gov/sites/default/files/recommended_practices/DHS_Common_Cybersecurity_Vulnerabilities_ICS_2010.pdf.

[98] U.S. Department of Homeland Security. *ICS-CERT Monitor September 2014 – February 2015*. 2015. URL: https://ics-cert.us-cert.gov/sites/default/files/Monitors/ICS-CERT_Monitor_Sep2014-Feb2015.pdf.

[99] IDAPython. *IDAPython*. 2017. URL: https://github.com/idapython/src.

[100] Illmatics. *Carhacking*. URL: http://illmatics.com/carhacking.html.

[101] Gartner Inc. *Gartner Identifies the Top 10 Internet of Things Technologies for 2017 and 2018*. 2016. URL: http://www.gartner.com/newsroom/id/3221818.

[102] Gartner Inc. *Gartner Says 6.4 Billion Connected 'Things' Will Be in Use in 2016, Up 30 Percent From 2015*. 2016. URL: http://www.gartner.com/newsroom/id/3165317.

[103] MIPS Technologies Inc. *MIPS32 Architecture For Programmers Volume III: The MIPS32 Privileged Resource Architecture*. URL: http://www.cs.cornell.edu/courses/cs3410/2015sp/MIPS_Vol3.pdf.

[104] Wind River Systems Inc. *Wind River to Acquire DSP Technology from Eonic Systems*. URL: https://www.windriver.com/news/press/pr.html?ID=437.

[105] x86 Instruction Set Reference. *RDTSC: Read Time-Stamp Counter*. URL: http://x86.renejeschke.de/html/file_module_x86_id_278.html.

[106] Texas Instruments. *LM3S6965 - Stellaris LM3S Microcontroller*. 2017. URL: http://www.ti.com/product/LM3S6965.

[107] Intel. "Chapter 17.4: Last Branch, Interrupt, and Exception Recording Overview." In: *Intel 64 and IA-32 architectures software developer's manual* 3 (2016). URL: https://software.intel.com/en-us/articles/intel-sdm.

[108] Intel. *Control-flow Enforcement Technology Preview*. 2016. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[109] Intel. *Intel Quark Microcontroller Software Interface Bootloader*. 2017. URL: http://www.intel.com/content/dam/www/public/us/en/documents/guides/software-interface-bootloader-user-guide.pdf.

[110] Nigel Jones. *Computing your stack size*. 2009. URL: http://embeddedgurus.com/stack-overflow/2009/03/computing-your-stack-size/.

[111] Peter Kacherginsky. *IDA Sploiter*. 2014. URL: http://thesprawl.org/projects/ida-sploiter/.

[112] Samy Kamkar. *phpwn: Attacking sessions and pseudo-random*. 2010. URL: https://samy.pl/phpwn/.

[113] Samy Kamkar. *SkyJack: autonomous drone hacking*. 2013. URL: http://samy.pl/skyjack/.

[114] Yaakov Katz. *IDF Encrypting Drones After Hizbullah Accessed Footage*. 2010. URL: http://www.jpost.com/Israel/IDF-encrypting-drones-after-Hizbullah-accessed-footage.

[115] Yongdae Kim. *Hacking Sensors*. 2017. URL: https://www.usenix.org/sites/default/files/conference/protected-files/enigma17_slides_kim.pdf.

[116] Tim King. *Migrating safety-critical software in military and aerospace systems*. 2011. URL: http://mil-embedded.com/articles/migrating-safety-critical-software-military-aerospace-systems/.

[117] Steve Klabnik. *intermezzOS, (a little OS)*. 2017. URL: http://intermezzos.github.io/.

[118] Tobias Klein. *checksec*. URL: http://www.trapkit.de/tools/checksec.html.

[119] Ronald van der Knijff. *Mobile and Embedded Device Forensics*. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.186.9014&rep=rep1&type=pdf.

[120] Ronald van der Knijff. "Control systems/SCADA forensics, what's the difference?" In: *Digital Investigation* (2014).

[121] Keith Kohl. *Enterprise Leads the 'Internet of Things' Growth*. 2015. URL: https://www.energyandcapital.com/articles/enterprise-leads-the-internet-of-things-growth/4919.

[122] Adrian Kosmaczewski. *The QNX Realtime Operating System*. 2007. URL: http://home.iitj.ac.in/~saurabh.heda/Papers/Survey/Report%20on%20%20QNX%20-2007.pdf.

[123] Paul Kot. *ROPC*. 2013. URL: https://github.com/pakt/ropc.

[124] Nick Kralevich. *How vulnerabilities help shape security features and mitigations in Android*. 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Kralevich-The-Art-Of-Defense-How-Vulnerabilities-Help-Shape-Security-Features-And-Mitigations-In-Android.pdf.

[125] Brian Krebs. *Target Hackers Broke in Via HVAC Company*. 2014. URL: https://krebsonsecurity.com/2014/02/target-hackers-broke-in-via-hvac-company/.

[126] Brian Krebs. *Hacked Cameras, DVRs Powered Today's Massive Internet Outage*. 2016. URL: https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/.

[127] Logan Lamb. *Home Insecurity: No Alarms, False Alarms, and SIGINT*. 2014. URL: https://suretydiy.com/wp-content/uploads/DEFCON-22-Logan-Lamb-HOME-INSECURITY-NO-ALARMS-FALSE-ALARMS-.pdf.

[128] John Lambert. *The Inside Story Behind MS08-067*. 2015. URL: https://blogs.technet.microsoft.com/johnla/2015/09/26/the-inside-story-behind-ms08-067/.

[129] Kristian Weium Lange. "Cybersecurity in the Internet of Things." In: (2016).

[130] Lennart Langenhop. *Embedded Security Analysis for an Engine Control Unit Architecture*. 2015. URL: http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lla-mt.pdf.

[131] Edward A. Lee. *Cyber Physical Systems: Design Challenges*. Tech. rep. UCB/EECS-2008-8. EECS Department, University of California, Berkeley, 2008. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html.

[132] Vincent van der Leest, Erik van der Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh. "Cryptography and Security." In: ed. by David Naccache. Berlin, Heidelberg: Springer-Verlag, 2012. Chap. Efficient Implementation of True Random Number Generator Based on SRAM PUFs, pp. 300–318. ISBN: 978-3-642-28367-3. URL: http://dl.acm.org/citation.cfm?id=2184081.2184106.

[133] John Leyden. *Entropy drought hits Raspberry Pi harvests, weakens SSH security*. 2015. URL: https://www.theregister.co.uk/2015/12/02/raspberry_pi_weak_ssh_keys/.

[134] Sung-Soo Lim. *SNU Real-time Benchmarks*. 2017. URL: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[135] Massachusetts Institute of Technology Lincoln Laboratory. *High Performance Embedded Computing (HPEC) Challenge Benchmark Suite*. 2017. URL: http://www.omgwiki.org/hpec/files/hpec-challenge/.

[136] Felix 'FX' Lindner. *Cisco IOS Router Exploitation: A map of the problem space*. 2009. URL: http://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-PAPER.pdf.

[137] Gentoo Linux. *Hardened/GNU stack quickstart*. URL: https://wiki.gentoo.org/wiki/Hardened/GNU_stack_quickstart.

[138] LittleBlackBox. URL: https://code.google.com/archive/p/littleblackbox/.

[139] DigiReach Pvt. Ltd. *Trends in the Internet of Things*. 2016. URL: http://www.digireach.com/trends-in-the-internet-of-things/.

[140] Michael Lynn. *The Holy Grail: Cisco IOS Shellcode And Exploitation Techniques*. URL: https://bernd-paysan.de/lynn-cisco.pdf.

[141] Mälardalen Real-Time Research Center (MRTC). *WCET project / Benchmarks*. 2017. URL: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[142] Microsoft MSDN. *Control Flow Guard*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[143] Microsoft MSDN. *CryptGenRandom*. 2017. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379942(v=vs.85).aspx.

[144] Dennis Maldonado. *Are We Really Safe? - Bypassing Access Control Systems*. 2015. URL: https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEFCON-23-Dennis-Maldonado-Are-we-really-safe-bypassing-access-control-systems-UPDATED.pdf.

[145] Marcus Hutchins (MalwareTech). *Mapping Mirai: A Botnet Case Study*. 2016. URL: https://www.malwaretech.com/2016/10/mapping-mirai-a-botnet-case-study.html.

[146] Tarjei Mandt. *Attacking the iOS Kernel: A Look at 'evasion'*. 2013. URL: http://blog.azimuthsecurity.com/2013/03/attacking-ios-kernel-look-at-evasi0n.html.

[147] Tarjei Mandt. "Revisiting iOS Kernel (In)Security: Attacking the early random() PRNG." In: *Azimuth Security* (2014). URL: http://mista.nu/research/early_random-paper.pdf.

[148] Daniel Micay. *The State of ASLR on Android Lollipop*. 2015. URL: https://copperhead.co/blog/2015/05/11/aslr-android-zygote.

[149] Microsemi. *Microsemi Secure Boot Reference Design*. 2014. URL: https://www.microsemi.com/document-portal/doc_view/133604-microsemi-secure-boot-reference-design-white-paper.

[150] Daniel Miessler. *IoT Attack Surface Mapping*. 2015. URL: https://www.rsaconference.com/writable/presentations/file_upload/asd-t10-securing-the-internet-of-things-mapping-iot-attack-surface-areas-with-the-owasp-iot-top-10-project.pdf.

[151] Matt Miller. "Reducing the Effective Entropy of GS Cookies." In: *Uninformed Vol. 7* (2007). URL: http://uninformed.org/?v=7&a=2&t=pdf.

[152] Ingo Molnar. *NX: clean up legacy binary support, 2.6.8-rc2*. 2004. URL: https://lwn.net/Articles/94068/.

[153] HD Moore. *Fun with VxWorks*. 2010. URL: https://speakerdeck.com/hdm/fun-with-vxworks.

[154] Tilo Muller. *ASLR Smack & Laugh Reference*. 2008. URL: https://ece.uwaterloo.ca/~vganesh/TEACHING/S2014/ECE458/aslr.pdf.

[155] NIST. *NIST Entropy Source Testing (EST) tool*. URL: https://github.com/usnistgov/SP800-90B_EntropyAssessment.

[156] NIST. *NIST Statistical Test Suite (STS)*. URL: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.

[157] NIST. "NIST SP800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications." In: *NIST* (2010). URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf.

[158] NIST. "NIST SP800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation." In: *NIST* (2016). URL: http://csrc.nist.gov/publications/drafts/800-90/sp800-90b_second_draft.pdf.

[159] NIST. *Lightweight Cryptography*. 2017. URL: https://www.nist.gov/programs-projects/lightweight-cryptography.

[160] STMicroelectronics N.V. *DM37051: ARM Cortex-M4 STM32F405xx STM32F407xx datasheet*. 2012. URL: http://www.st.com/resource/en/datasheet/stm32f407vg.pdf.

[161] STMicroelectronics N.V. *DM88500: STM32F030x4 STM32F030x6 STM32F030x8 STM32F030xC datasheet*. 2015. URL: http://www.st.com/resource/en/datasheet/stm32f030c8.pdf.

[162] NXP. *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*. URL: http://www.nxp.com/assets/documents/data/en/reference-manuals/MPCFPE32B.pdf.

[163] Anas Nashif. *The Zephyr Project: Reflection on the First Year and Plans for the Next Year*. 2017. URL: https://schd.ws/hosted_files/openiotelcna2017/99/zephyr_openiot_na_2017.pdf.

[164] Anas Nashif. *Zephyr Project Overview*. 2017. URL: http://events.linuxfoundation.org/sites/events/files/slides/Zephyr%20Overview%20-%20OpenIOT%20Summit%202016.pdf.

[165] NetMarketShare. *Mobile/Tablet Operating System Market Share*. URL: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1.

[166] Tuan Nguyen. *How college students hijacked a government spy drone*. 2012. URL: http://www.zdnet.com/article/how-college-students-hijacked-a-government-spy-drone/.

[167] Bojan Nikolic. *The LD_DEBUG environment variable*. URL: http://www.bnikolic.co.uk/blog/linux-ld-debug.html.

[168] Karsten Nohl. *Car immobilizer hacking*. 2013. URL: https://media.ccc.de/v/konferenz_mp6_og_-_2013-07-05_17:00_-_car_immobilizer_hacking_-_karsten_nohl_-_5034.

[169] Karsten Nohl. *Corroding immobilizer cryptography*. 2013. URL: http://www.nosuchcon.org/talks/2013/D2_05_KNohl_Immobilizer_Security.pdf.

[170] Colin O'Flynn. *Dont Whisper my Chips: Sidechannel and Glitching for Fun and Profit*. 2015. URL: https://www.youtube.com/watch?v=BHqrA8lzz2o.

[171] OWASP. *C-Based Toolchain Hardening*. 2017. URL: https://www.owasp.org/index.php/C-Based_Toolchain_Hardening.

[172] Jon Oberheide. *Analyzing ASLR in Android Ice Cream Sandwich 4.0*. 2012. URL: https://threatpost.com/analyzing-aslr-android-ice-cream-sandwich-40-022112/76239/.

[173] Nancy Owano. *RQ-170 drone's ambush facts spilled by Iranian engineer*. 2011. URL: https://phys.org/news/2011-12-rq-drone-ambush-facts-iranian.html.

[174] PaX. *NOEXEC*. URL: https://pax.grsecurity.net/docs/noexec.txt.

[175] FreeBSD Man Pages. *RANDOM(3)*. URL: https://www.freebsd.org/cgi/man.cgi?query=random&sektion=3.

[176] Niheer Patel. *Wind River Welcomes Linux Foundation's Zephyr Project*. URL: http://blogs.windriver.com/wind_river_blog/2016/02/wind-river-welcomes-linux-foundations-zephyr-project.html.

[177] Kasper Pedersen. *Entropy gathering for cryptographic applications in AVR - Qualification of WDT as entropy source*. 2002. URL: http://n1.taur.dk/avrfreaks/engather.pdf.

[178] Dale Peterson. *Project Basecamp Foreshadows Ukraine Bad Firmware Upload*. 2016. URL: http://www.digitalbond.com/blog/2016/03/21/project-basecamp-foreshadows-ukraine-bad-firmware-upload/.

[179] Alex Plaskett. *QNX Security Architecture*. 2016. URL: https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-qnx-security-whitepaper-2016-03-14.pdf.

[180] Erik Poll. "Safe programming languages." In: *Lecture Notes on Language-Based Security* (2016). URL: https://www.cs.ru.nl/E.Poll/papers/language_based_security.pdf.

[181] IDA Pro. *IDA Pro*. 2017. URL: https://www.hex-rays.com/products/ida/.

[182] LLVM Project. *MachineRegisterInfo.cpp*. 2017. URL: https://github.com/llvm-mirror/llvm/blob/05383dbf2bb4f3fb40a347cf83121dea848da7
lib/CodeGen/MachineRegisterInfo.cpp.

[183] LLVM Project. *The LLVM Compiler Infrastructure*. 2017. URL: https://llvm.org/.

[184] LLVM Project. *Writing an LLVM Pass*. 2017. URL: http://llvm.org/docs/WritingAnLLVMPass.html.

[185] LLVM Project. *clang: a C language family frontend for LLVM*. 2017. URL: https://clang.llvm.org/.

[186] Zephyr Project. *Zephyr*. URL: https://www.zephyrproject.org/.

[187] Max Thomas (Shiny Quagsire). *3DS Userland Exploit for Pokemon Super Mystery Dungeon*. 2017. URL: https://github.com/shinyquagsire23/supermysterychunkhax.

[188] Open Source RTOS. *List of open source real-time operating systems*. 2017. URL: https://www.osrtos.com/.

[189] Jerome Radcliffe. *Hacking Medical Devices for Fun and Insulin: Breaking the Human. SCADA System*. 2011. URL: https://media.blackhat.com/bh-us-11/Radcliffe/BH_US_11_Radcliffe_Hacking_Medical_Devices_WP.pdf.

[190] Paul Rascagneres. *Stack Smashing Protector*. 2010. URL: http://www.hackitoergosum.org/2010/HES2010-prascagneres-Stack-Smashing-Protector-in-FreeBSD.pdf.

[191] Ray. *Lockpicking in the IoT*. 2016. URL: https://fahrplan.events.ccc.de/congress/2016/Fahrplan/system/event_attachments/attachments/000/003/115/original/33c3-ray-lockpicking-in-the-IoT.pdf.

[192] James Reeds. *'Cracking' A Random Number Generator*. 1977. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.6089&rep=rep1&type=pdf.

[193] Jason Reeves. *Autoscopy Jr.: Intrusion Detection for Embedded Control Systems*. 2011. URL: https://pdfs.semanticscholar.org/e654/20dae3e41d85d0f09e0049ae5ba7078b4a86.pdf.

[194] Gerardo Richarte. *Four different tricks to bypass StackShield and StackGuard protection*. 2002. URL: http://staff.ustc.edu.cn/~bjhua/courses/security/2014/readings/stackguard-bypass.pdf.

[195] Billy Rios. *Pulling the Curtain on Airport Security*. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Rios-Pulling-Back-The-Curtain-On-Airport-Security.pdf.

[196] Nils Rodday. *Hacking a Professional Drone*. 2016. URL: https://www.blackhat.com/docs/asia-16/materials/asia-16-Rodday-Hacking-A-Professional-Drone.pdf.

[197] Dan Rosenberg. *SMEP: What is It, and How to Beat It on Linux*. 2011. URL: http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/.

[198] STMicroelectronics. *SH-4, ST40 System Architecture, Volume 1: System*. URL: http://www.st.com/content/ccc/resource/technical/document/reference_manual/b1/d6/c7/6d/13/97/4c/30/CD17153464.pdf/files/CD17153464.pdf/jcr:content/translations/en.CD17153464.pdf.

[199] Jonathan Salwan. *ROPgadget*. 2017. URL: https://github.com/JonathanSalwan/ROPgadget/.

[200] Miro Samek. *Are We Shooting Ourselves in the Foot with Stack Overflow?* 2014. URL: http://embeddedgurus.com/state-space/2014/02/are-we-shooting-ourselves-in-the-foot-with-stack-overflow/.

[201] Ruben Santamarta. *SATCOM Terminals: Hacking by Air, Sea, and Land*. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Santamarta-SATCOM-Terminals-Hacking-By-Air-Sea-And-Land-WP.pdf.

[202] Bruce Schneier. "The Internet of Things Is Wildly Insecure - And Often Unpatchable." In: *Wired* (2014). URL: https://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem/.

[203] Bruce Schneier. *Security and the Internet of Things*. 2017. URL: https://www.schneier.com/blog/archives/2017/02/security_and_th.html.

[204] Security and University of Luxembourg Trust Center CSC Research Unit. *Lightweight Cryptography*. 2017. URL: https://www.cryptolux.org/index.php/Lightweight_Cryptography.

[205] Senrio. *The Insecurity Of Things*. 2016. URL: http://blog.senr.io/blog/the-insecurity-of-things.

[206] Fermin J. Serna. "The info leak era on software exploitation." In: *Black Hat US* (2012). URL: https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf.

[207] Chris Simmonds. *Software update for IoT: the current state of play*. 2016. URL: http://events.linuxfoundation.org/sites/events/files/slides/software-update-elce-2016-169.pdf.

[208] Doug Simon. *Stacks with split personalities*. 2017. URL: https://blogs.oracle.com/dns/entry/stacks_with_split_personalities.

[209] Tom Simon. *Processors Rule the Day*. 2015. URL: https://www.semiwiki.com/forum/content/5086-processors-rule-day.html.

[210] Jordan Rabet (Smealum). *limited physical layout randomization for Nintendo 3DS 10.4*. 2017. URL: https://twitter.com/smealum/status/689688615627030528.

[211] Jordan Rabet (Smealum). *source code for 3DS hax 2.x payloads*. 2017. URL: https://github.com/smealum/ninjhax2.x.

[212] Craig Smith. *The Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press, 2016.

[213] Rini van Solingen. "Integrating Software Engineering Technologies For Embedded Systems Development." In: *PROFES* (2002).

[214] QNX Software Systems. *ClockCycles()*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fc%2Fclockcycles.html.

[215] QNX Software Systems. *ClockTime(), ClockTime_r()*. URL: http://www.qnx.com/developers/docs/660/topic/com.qnx.doc.neutrino.lib_ref/topic/c/clocktime.html.

[216] QNX Software Systems. *Controlling processes via the /proc filesystem*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fprocess_proc_filesystem.html.

[217] QNX Software Systems. *DCMD_PROC_INFO*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.cookbook%2Ftopic%2Fs3_procfs_DCMD_PROC_INFO.html.

[218] QNX Software Systems. *Memory management in QNX Neutrino*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fmemory_MEMMGT_.html.

[219] QNX Software Systems. *On*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities%2Ftopic%2Fo%2Fon.html.

[220] QNX Software Systems. *Private virtual memory*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fproc_Private_virtual_memory.html.

[221] QNX Software Systems. *Procnto*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities/topic/p/procnto.html.

[222] QNX Software Systems. *QNX Markets: Defense*. URL: http://www.qnx.com/content/qnx/en/solutions/industries/defense/index.html.

[223] QNX Software Systems. *QNX Markets: Industrial*. URL: http://www.qnx.com/content/qnx/en/solutions/industries/automation/index.html.

[224] QNX Software Systems. *QNX Markets: Medical*. URL: http://www.qnx.com/content/qnx/en/solutions/industries/medical/index.html.

[225] QNX Software Systems. *QNX Markets: Rail Safety*. URL: http://www.qnx.com/content/qnx/en/solutions/industries/rail-safety/index.html.

[226] QNX Software Systems. *QNX Momentics Development Suite 6.3.2 Release Notes*. URL: http://www.qnx.com/developers/docs/6.3.2/momentics/release_notes/rel_6.3.2.html.

[227] QNX Software Systems. *QNX Momentics Tool Suite*. URL: http://www.qnx.com/content/qnx/en/products/tools/qnx-momentics.html.

[228] QNX Software Systems. *Random*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities/topic/r/random.html.

[229] QNX Software Systems. *devctl*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fd%2Fdevctl.html.

[230] QNX Software Systems. *pidin*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=/com.qnx.doc.neutrino.utilities/topic/p/pidin.html.

[231] QNX Software Systems. *posix_spawn*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fposix_spawn.html.

[232] QNX Software Systems. *spawn*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fproc_spawn.html.

[233] QNX Software Systems. *QNX Neutrino RTOS: System Architecture*. 2014. URL: http://support7.qnx.com/download/download/26183/QNX_Neutrino_RTOS_System_Architecture.pdf.

[234] QNX Software Systems. *50 Million Vehicles and Counting: QNX Achieves New Milestone in Automotive Market*. 2015. URL: http://www.qnx.com/news/pr_6118_3.html.

[235] Jack Tang. *Exploring Control Flow Guard in Windows 10*. URL: https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf.

[236] Chris Tarnovsky. *Semiconductor Security Awareness, Today and Yesterday*. 2010. URL: https://www.youtube.com/watch?v=WXX00tRKOlw.

[237] SolidState Technology. *Comparing market sizes and forecasted growth rates for systems, ICs*. 2015. URL: http://electroiq.com/blog/2015/03/comparing-market-sizes-and-forecasted-growth-rates-for-systems-ics/.

[238] Julien Tinnes. *Local bypass of Linux ASLR through /proc information leaks*. 2009. URL: http://blog.cr0.org/2009/04/local-bypass-of-linux-aslr-through-proc.html.

[239] *Tock OS*. URL: https://www.tockos.org/.

[240] UBM. *UBM Embedded Markets Study 2013*. 2013. URL: http://bd.eduweb.hhs.nl/es/2013_Embedded_Market_Study_Final.pdf.

[241] UBM. *UBM Embedded Markets Study 2014*. 2014. URL: http://bd.eduweb.hhs.nl/es/2014-embedded-market-study-then-now-whats-next.pdf.

[242] UBM. *UBM Embedded Markets Study 2015*. 2015. URL: https://webpages.uncc.edu/~jmconrad/ECGR4101-2015-08/Notes/UBM%20Tech%202015%20Presentation%20of%20Embedded%20Markets%20Study%20World%20Day1.pdf.

[243] Ubuntu. *ld.so, ld-linux.so - dynamic linker/loader*. 2017. URL: http://manpages.ubuntu.com/manpages/xenial/man8/ld.so.8.html.

[244] Ubuntu. */proc/pid/maps protection*. 2017. URL: https://wiki.ubuntu.com/Security/Features#proc-maps.

[245] Verizon. "State of the Market: The Internet of Things 2015." In: (2015). URL: http://www.verizonenterprise.com/resources/reports/rp_state-of-market-the-market-the-internet-of-things-2015_en_xg.pdf.

[246] Job Vranish. *Using Rust 1.8 Stable for Building Embedded Firmware*. 2016. URL: https://spin.atomicobject.com/2016/05/25/rust-1-8-embedded-firmware/.

[247] Ralf-Philipp Weinmann. "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks." In: *WOOT* (2012). URL: https://www.usenix.org/system/files/conference/woot12/woot12-final24.pdf.

[248] Ralf-Philipp Weinmann. *BlackberryOS 10 From a Security Perspective*. 2013. URL: https://www.youtube.com/watch?v=eCbJ40fbJr4.

[249] Victor Wen. "Security on Tire Pressure Monitor System." In: (2005). URL: http://bwrcs.eecs.berkeley.edu/Classes/icdesign/ee241_s05/Projects/Midterm/VictorWen.pdf.

[250] David Weston. *Windows 10 Mitigation Improvements*. 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf.

[251] Jos Wetzels. "Broken keys to the kingdom: Security and privacy aspects of RFID-based car keys." In: (2014). URL: https://arxiv.org/ftp/arxiv/papers/1405/1405.7424.pdf.

[252] Jos Wetzels. *Cryptanalysis of intercepted Israeli drone feeds*. 2016. URL: http://samvartaka.github.io/cryptanalysis/2016/02/02/videocrypt-uavs.

[253] WikiLeaks. *Vault 7: CIA Hacking Tools Revealed - 2014-10-23 Branch Direction Meeting notes*. 2017. URL: https://wikileaks.org/ciav7p1/cms/page_13763790.html.

[254] Wikipedia. *Kintsugi*. URL: https://en.wikipedia.org/wiki/Kintsugi.

[255] Wikipedia. *Min entropy*. URL: https://en.wikipedia.org/wiki/Min_entropy.

[256] Wikipedia. *Rényi entropy*. URL: https://en.wikipedia.org/wiki/RÃl’nyi_entropy.

[257] Wikipedia. *ACPI*. 2017. URL: https://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface.

[258] Wikipedia. *ASR9000*. 2017. URL: https://en.wikipedia.org/wiki/ASR9000.

[259] Wikipedia. *Carrier Routing System*. 2017. URL: https://en.wikipedia.org/wiki/Carrier_Routing_System.

[260] Wikipedia. *Cisco 12000*. 2017. URL: https://en.wikipedia.org/wiki/Cisco_12000.

[261] Wikipedia. *Homebrew (video games)*. 2017. URL: https://en.wikipedia.org/wiki/Homebrew_(video_games).

[262] Wikipedia. *IOS XR*. 2017. URL: https://en.wikipedia.org/wiki/IOS_XR.

[263] Wikipedia. *Intel MPX*. 2017. URL: https://en.wikipedia.org/wiki/Intel_MPX.

[264] Wikipedia. *Intel SGX*. 2017. URL: https://software.intel.com/en-us/sgx.

[265] Wikipedia. *Translation lookaside buffer*. 2017. URL: https://en.wikipedia.org/wiki/Translation_lookaside_buffer.

[266] Wikipedia. *Trusted Execution Technology*. 2017. URL: https://en.wikipedia.org/wiki/Trusted_Execution_Technology.

[267] Wikipedia. *procfs*. 2017. URL: https://en.wikipedia.org/wiki/Procfs.

[268] Roy S. Wikramaratna. "The additive congruential random number generator - a special case of a multiple recursive generator." In: *Journal of Computational and Applied Mathematics* (2008).

[269] Ramesh Yerraballi. "Real-Time Operating Systems: An Ongoing Review." In: *RTSSWIP* (2000).

[270] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd edition*. Elsevier Science & Technology, 2013.

[271] Yang Yu. *DEP/ASLR bypass without ROP/JIT*. 2013. URL: https://cansecwest.com/slides/2013/DEP-ASLR%20bypass%20without%20ROP-JIT.pdf.

[272] Zhang Yunhai. *Bypass Control Flow Guard Comprehensively*. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf.

[273] Adam 'pi3'Zabrocki. "Scraps of notes on remote stack overflow exploitation." In: *Phrack* (2010). URL: http://phrack.org/issues/67/13.html.

[274] Zephyr. *Dining Philosophers*. 2017. URL: https://www.zephyrproject.org/doc/samples/philosophers/README.html.

[275] Zephyr. *Echo Server*. 2017. URL: https://www.zephyrproject.org/doc/samples/net/echo_server/README.html.

[276] Zephyr. *Sample TELNET console application*. 2017. URL: https://www.zephyrproject.org/doc/samples/net/telnet/README.html.

[277] Kim Zetter. *A Cyberattack Has Caused Confirmed Physical Damage for the Second Time Ever*. 2015. URL: https://www.wired.com/2015/01/german-steel-mill-hack-destruction/.

[278] Kim Zetter. *The Ukrainian Power Grid Was Hacked Again*. 2017. URL: http://motherboard.vice.com/read/ukrainian-power-station-hacking-december-2016-report.

[279] Dino A. Dai Zovi. *Practical Return-Oriented Programming*. 2010. URL: https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

[280] Dino A. Dai Zovi. *Apple iOS 4 Security Evaluation*. 2011. URL: https://media.blackhat.com/bh-us-11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf.

[281] Abdulmalik Humayed et al. "Cyber-Physical Systems Security – A Survey." In: *IEEE Internet of Things Journal* (2017).

[282] Abid Malik et al. "Optimal basic block instruction scheduling for Multiple-Issue processors using constraint programming." In: *ICTAI* (2006).

[283] Adam Laurie et al. *Decapping Chips the Easy Hard Way*. 2013. URL: https://www.youtube.com/watch?v=vbIJ-eVQkaw.

[284] Aki Koivu et al. "Software Security Considerations for IoT." In: *iThings* (2016).

[285] Alex Plaskett et al. *QNX: 99 Problems but a Microkernel ain't one!* 2016. URL: https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-qnx-troopers-99-problems-but-a-microkernel-aint-one.pdf.

[286] Alexander Bolshev et al. *DTM components: shadow keys to the ICS kingdom*. 2014. URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Bolshev-DTM-Components-Shadow-Keys-To-The-ICS-Kingdom-wp.pdf.

[287] Alexander Bolshev et al. *ICSCorsair: How I will PWN your ERP through 4-20 mA current loop*. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Bolshev-ICSCorsair-How-I-Will-PWN-Your-ERP-Through-4-20mA-Current-Loop-WP.pdf.

[288] Alexander Bolshev et al. *Practical Firmware Reversing and Exploit Development for AVR-based Embedded Devices*. 2015. URL: http://2015.zeronights.org/assets/files/43-bolshev-ryutin.pdf.

[289] Ali Abbasi et al. *Ghost in the PLC: Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack*. 2016.

[290] Amit Levy et al. "Ownership Is Theft: Experiences Building an Embedded OS in Rust." In: *PLOS* (2015).

[291] Andreas Elvstam et al. "Operating systems for resource constraint Internet of Things devices: An evaluation." In: (2016).

[292] Andrei Homescu et al. "Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming." In: *WOOT* (2012). URL: https://www.sba-research.org/wp-content/uploads/publications/woot12.pdf.

[293] Andrei Homescu et al. "Profile-guided automated software diversity." In: *International Symposium on Code Generation and Optimization* (2013).

[294] Andrew J. Kerns et al. "Unmanned Aircraft Capture and Control Via GPS Spoofing." In: *Journal of Field Robotics* (2014).

[295] Andrey Sidorenko et al. *State Recovery Attacks on Pseudorandom Generators*. 2005. URL: http://www.win.tue.nl/~berry/papers/weworc05sra.pdf.

[296] Ang Cui et al. "Killing the Myth of Cisco IOS Diversity: Recent Advances in Reliable Shellcode Design." In: *WOOT* (2011). URL: https://www.usenix.org/legacy/event/woot11/tech/final_files/Cui.pdf.

[297] Anthony Rose et al. *Picking Bluetooth Low Energy Locks From a Quarter Mile Away*. 2016. URL: https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Rose-Ramsey-Picking-Bluetooth-Low-Energy-Locks-UPDATED.pdf.

[298] Anthony Van Herrewege et al. "Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers." In: *TrustED* (2013).

[299] Anthony Van Herrewege et al. "Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M." In: *DAC* (2014).

[300] Arijit Ukil et al. "Embedded Security for Internet of Things." In: *NCETACS* (2011).

[301] Arnaud Lebrun et al. *CANSPY: a Platform for Auditing CAN Devices*. 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Demay-CANSPY-A-Platorm-For-Auditing-CAN-Devices-wp.pdf.

[302] Aurélien Francillon et al. "TinyRNG: A Cryptographic Random Number Generator for Wireless Sensors Network Nodes." In: *WiOpt* (2007).

[303] Aurélien Francillon et al. "Code injection attacks on harvard-architecture devices." In: *CCS* (2009).

[304] Ben Lampert et al. "Robust, low-cost, auditable random number generation for embedded system security." In: *SenSys* (2016).

[305] Benedikt Kristinsson et al. “Ardrand: The Arduino as a Hardware Random-Number Generator.” In: (2012). URL: https://skemman.is/bitstream/1946/10689/1/ardrand.pdf.

[306] Byoungyoung Lee et al. “From Zygote to Morula: Fortifying Weakened ASLR on Android.” In: *Security & Privacy* (2014).

[307] Charlie Miller et al. *iOS Hacker's Handbook*. Wiley, 2012.

[308] Charlie Miller et al. “Remote Exploitation of an Unaltered Passenger Vehicle.” In: (2015). URL: http://illmatics.com/Remote%20Car%20Hacking.pdf.

[309] Charlie Miller et al. “CAN Message Injection.” In: (2016). URL: http://illmatics.com/can%20message%20injection.pdf.

[310] Charlie Miller et al. “Car Hacking: For Poories a.k.a. Car Hacking Too: Electric Boogaloo.” In: (2017). URL: http://illmatics.com/car_hacking_poories.pdf.

[311] Chris Valasek et al. “Adventures in Automotive Networks and Control Units.” In: (2014). URL: https://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf.

[312] Christian Collberg et al. “A taxonomy of obfuscating transformations.” In: *Computer Science Technical Reports 148* (1997).

[313] Christine Hennebert et al. “Entropy Harvesting from Physical Sensors.” In: *WiSec* (2013).

[314] Chunxiao Li et al. “Hijacking an Insulin Pump: Security Attacks and Defenses for a Diabetes Therapy System.” In: *Healthcom* (2011).

[315] Cliff Young et al. “Near-optimal intraprocedural branch alignment.” In: *SIGPLAN* (1997).

[316] Cor Meenderinck et al. “Composable Virtual Memory for an Embedded SoC.” In: *DSD* (2012).

[317] Cristiano Giuffrida et al. “Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization.” In: *USENIX Security* (2012).

[318] Daniel Crowley et al. *Home Invasion 2.0: Attacking Network-Connected Embedded Devices*. 2013. URL: https://media.blackhat.com/us-13/US-13-Crowley-Home-Invasion-2-0-WP.pdf.

[319] Daniel E. Holcomb et al. “Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags.” In: *RFIDsec* (2007).

[320] Daniel Halperin et al. “Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses.” In: *Security & Privacy* (2008).

[321] Daniel J. Bernstein et al. “Factoring RSA keys from certified smart cards: Coppersmith in the wild.” In: *ASIACRYPT* (2013).

[322] Daniel Martin Gomez et al. *BlackBerry PlayBook Security: Part one*. 2011. URL: https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/blackberry_playbook_security._part_one.pdf.

[323] Daniel Trivellato et al. “Lights out! Who’s next? Analysis and detection of the Ukrainian ’cyber-blackout’.” In: (2016).

[324] David Barksdale et al. *VxWorks: Execute My Packets*. 2016. URL: https://blog.exodusintel.com/2016/08/09/vxworks-execute-my-packets/.

[325] David Formby et al. “Out of Control: Ransomware for Industrial Control Systems.” In: (2017). URL: http://www.cap.gatech.edu/plcransomware.pdf.

[326] David Kaplan et al. “Attacking the Linux PRNG On Android: Weaknesses in Seeding of Entropic Pools and Low Boot-Time Entropy.” In: *WOOT* (2014). URL: https://www.usenix.org/system/files/conference/woot14/woot14-kaplan.pdf.

[327] David Kaplan et al. “Attacking the Linux PRNG on Android & Embedded Devices.” In: *Black Hat Europe* (2014). URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Kedmi-Attacking-The-Linux-PRNG-On-Android-Weaknesses-In-Seeding-Of-Entropic-Pools-And-Low-Boot-Time-Entropy.pdf.

[328] Denis Foo Kune et al. “Ghost Talk: Mitigating EMI Signal Injection Attacks against Analog Sensors.” In: *Security & Privacy* (2013).

[329] Dennis K. Nilsson et al. “A Framework for Self-Verification of Firmware Updates over the Air in Vehicle ECUs.” In: *GLOBECOM* (2008).

[330] Dominik Lang et al. “Security Evolution in Vehicular Systems.” In: (2016). URL: https://edoc.hu-berlin.de/bitstream/handle/18452/2085/lang.pdf?sequence=1&isAllowed=y.

[331] Eduardo Novella Lorente et al. “Scrutinizing WPA2 Password Generating Algorithms in Wireless Routers.” In: *WOOT* (2015). URL: https://www.usenix.org/system/files/conference/woot15/woot15-paper-lorente.pdf.

[332] Edward J. Schwartz et al. “Q: exploit hardening made easy.” In: *USENIX Security* (2011).

[333] Edward J.M. Colbert et al. “Intrusion Detection in Industrial Control Systems.” In: *Cyber-security of SCADA and Other Industrial Control Systems* (2016).

[334] Elena Andreeva et al. "Security of Keyed Sponge Constructions Using a Modular Proof Approach." In: *FSE* (2015).

[335] Eric D. Knapp et al. *Applied Cyber Security and the Smart Grid*. Syngress, 2013.

[336] Erik Bosman et al. "Framing Signals - A Return to Portable Shellcode." In: *Security & Privacy* (2014).

[337] Erik Buchanan et al. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC." In: *CCS* (2008).

[338] Fardin Abdi Taghi Abad et al. "On-Chip Control Flow Integrity Check for Real Time Embedded Systems." In: *CPSNA* (2013).

[339] Felix Schuster et al. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *Security & Privacy* (2015).

[340] Fengwei Zhang et al. "A Framework to Secure Peripherals at Runtime." In: *ESORICS* (2014).

[341] Flavio D. Garcia et al. "Lock It and Still Lose It - on the (In)Security of Automotive Remote Keyless Entry Systems." In: *USENIX Security* (2016).

[342] Florian Kohnhäuser et al. "PUF-based Software Protection for Low-end Embedded Devices." In: *TRUST* (2015).

[343] Frank Adelstein et al. "Malicious Code Detection for Open Firmware." In: *ACSAC* (2002).

[344] Franz Rammig et al. "Basic Concepts of Real Time Operating Systems." In: *Hardware-dependent Software: Principles and Practice* (2009).

[345] Gabriel Pedroza et al. "A Formal Methodology Applied to Secure Over-the-Air Automotive Applications." In: *VTC* (2011).

[346] Gaurav S. Kc et al. "Countering code-injection attacks with instructionset randomization." In: *ACM Conference on Computer and Communications Security* (2003).

[347] George Argyros et al. *PRNG: Pwning Random Number Generators*. 2012. URL: https://media.blackhat.com/bh-us-12/Briefings/Argyros/BH_US_12_Argyros_PRNG_WP.pdf.

[348] Giuseppe Lo Re et al. "Secure random number generation in wireless sensor networks." In: *SIN* (2011).

[349] Grant Hernandez et al. *Smart Nest Thermostat: A Smart Spy in Your Home*. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Jin-Smart-Nest-Thermostat-A-Smart-Spy-In-Your-Home-WP.pdf.

[350] Guido Bertoni et al. "Sponge-based pseudo-random number generators." In: *CHES* (2010).

[351] Guido Bertoni et al. *The Keccak Reference*. 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[352] Guido Bertoni et al. "The Making of KECCAK." In: *Cryptologia* (2014).

[353] Gunnar Alendal et al. "got HW crypto? On the (in)security of a Self-Encrypting Drive series." In: *Cryptology ePrint Archive* (2015). URL: https://eprint.iacr.org/2015/1002.pdf.

[354] Hector Marco-Gisbert et al. "Preventing brute force attacks against stack canary protection on networking servers." In: *12th IEEE International Symposium on Network Computing and Applications (NCA)* (2013).

[355] Hector Marco-Gisbert et al. "On the Effectiveness of Full-ASLR on 64-bit Linux." In: *DeepSec* (2014).

[356] Hector Marco-Gisbert et al. "Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems." In: *BlackHat Asia* (2016). URL: https://www.blackhat.com/docs/asia-16/materials/asia-16-Marco-Gisbert-Exploiting-Linux-And-PaX-ASLRS-Weaknesses-On-32-And-64-Bit-Systems-wp.pdf.

[357] Heiko Falk et al. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research." In: *16th International Workshop on Worst-Case Execution Time Analysis* (2016).

[358] Houbing Song et al. *Smart Cities: Foundations, Principles and Applications*. Wiley, 2017.

[359] Hovav Shacham et al. "On the effectiveness of address-space randomization." In: *CCS* (2004).

[360] Hristo Bojinov et al. "Address Space Randomization for Mobile Devices." In: *WISEC* (2011).

[361] Ian Foster et al. "Fast and Vulnerable: A Story of Telematic Failures." In: *WOOT* (2015). URL: https://www.usenix.org/system/files/conference/woot15/woot15-paper-foster.pdf.

[362] Ian Moir et al. *Military Avionics Systems*. Wiley, 2006.

[363] Ian Moir et al. *Civil Avionics Systems, 2nd Ed.* Wiley, 2013.

[364] Insup Lee et al. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 2007.

[365] Isabelle Puaut et al. "Predictable Paging in Real-Time Systems: A Compiler Approach." In: *ECRTS* (2007).

[366] Ishtiaq Rouf et al. "Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study." In: *USENIX Security* (2010).

[367] Jan Werner et al. "No-Execute-After-Read: Preventing Code Disclosure in Commodity Software." In: *ASIACCS* (2016).

[368] Jannik Pewny et al. "Control-Flow Restrictor: Compiler-based CFI for iOS." In: *ACSAC* (2013).

[369] Jason Gionta et al. "Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification." In: *CNS* (2016).

[370] Jasper van Woudenberg et al. *20 ways past secure boot*. 2013. URL: https://conference.hitb.org/hitbsecconf2013kul/materials/D2T3%20-%20Job%20de%20Haas%20-%2020%20Ways%20Past%20Secure%20Boot.pdf.

[371] Javid Habibi et al. "MAVR: Code Reuse Stealthy Attacks and Mitigation on Unmanned Aerial Vehicles." In: *ICDCS* (2015).

[372] Jean-Marie Borello et al. "Code obfuscation techniques for metamorphic viruses." In: *Journal in Computer Virology* (2008).

[373] John Kelsey et al. "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator." In: *Sixth Annual Workshop on Selected Areas in Cryptography* (1999).

[374] Jonathan P. Bowen et al. "Safety-critical systems, formal methods and standards." In: *IEEE Software Engineering Journal* (1993).

[375] Jorge Guajardo et al. "FPGA intrinsic PUFs and their use for IP protection." In: *CHES* (2007).

[376] Jos Wetzels et al. *uShield: Host-based detection for embedded devices used in ICS environments*. URL: https://github.com/preemptive-FP7/uShield.

[377] Josef Hlaváč et al. "True Random Number Generation on an Atmel AVR Microcontroller." In: *ICCET* (2010).

[378] Joshua J. Drake et al. *Android Hacker's Handbook*. Wiley, 2014.

[379] Karl Koscher et al. "Experimental Security Analysis of a Modern Automobile." In: *Security & Privacy* (2010).

[380] Keaton Mowery et al. "Welcome to the Entropics: Boot-Time Entropy in Embedded Devices." In: *IEEE Security and Privacy* (2013).

[381] Ken Johnson et al. *Exploit Mitigation Improvements in Windows 8*. 2012. URL: https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf.

[382] Kim Hartmann et al. "UAV Exploitation: A New Domain for Cyber Power." In: *CyCon* (2016).

[383] Kjell Braden et al. "Leakage-Resilient Layout Randomization for Mobile Devices." In: *NDSS* (2016).

[384] Kyle Wallace et al. “Toward Sensor-Based Random Number Generation for Mobile and IoT Devices.” In: *IEEE Internet of Things Journal* (2016).

[385] Le Guan et al. “From Physical to Cyber: Escalating Protection for Personalized Auto Insurance.” In: *SenSys* (2016).

[386] Leyla Bilge et al. “Before we knew it: An empirical study of zero-day attacks in the real world.” In: *CCS* (2012).

[387] Liang Chen et al. *Subverting Apple Graphics: Practical Approaches to Remotely Gaining Root*. 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Chen-Subverting-Apple-Graphics-Practical-Approaches-To-Remotely-Gaining-Root.pdf.

[388] Lillian Ablon et al. “Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits.” In: *RAND Corporation* (2017). URL: http://www.rand.org/content/dam/rand/pubs/research_reports/RR1700/RR1751/RAND_RR1751.pdf.

[389] Loic Duflot et al. “What if you can’t trust your network card?” In: *RAID* (2011).

[390] László Szekeres et al. “SoK: Eternal War in Memory.” In: *Security and Privacy* (2013).

[391] Lucas Davi et al. “MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones.” In: *NDSS* (2012).

[392] Lucas Davi et al. “HAFIX: Hardware-Assisted Flow Integrity Extension.” In: *DAC* (2015).

[393] Luis A. Garcia et al. “Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit.” In: *NDSS* (2017).

[394] Mahmut Kandemir et al. “Compiler-Directed Code Restructuring for Reducing Data TLB Energy.” In: *CODES* (2004).

[395] Marc Dacier et al. *Network Attack Detection and Defense: Securing Industrial Control Systems for Critical Infrastructures*. 2014. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4791/pdf/dagrep_v004_i007_p062_s14292.pdf.

[396] Marcin Piotr Pawlowski et al. “Harvesting Entropy for Random Number Generation for Internet of Things Constrained Devices Using On-Board Sensors.” In: *Sensors* (2015).

[397] Marco Caselli et al. “Sequence-aware Intrusion Detection in Industrial Control Systems.” In: *CPSS* (2015).

[398] Mark Dowd et al. *iOS 6 Kernel Security: A Hacker’s Guide*. 2015. URL: http://conference.hackinthebox.org/hitbsecconf2012kul/materials/D1T2%20-%20Mark%20Dowd%20&%20Tarjei%20Mandt%20-%20iOS6%20Security.pdf.

[399] Matthew Guthaus et al. *MiBench*. 2017. URL: http://vhosts.eecs.umich.edu/mibench/.

[400] Maxim Chernyshev et al. “Security assessment of IoT devices: The case of two smart TVs.” In: *Australian Digital Forensics Conference* (2015).

[401] Michael Backes et al. “You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code.” In: *CCS* (2014).

[402] Michael Barr et al. *Embedded Systems Dictionary*. CRC Press, 2003.

[403] Mikhail Platonov et al. “Using Power-up SRAM State of Atmel ATmega1284P Microcontrollers as Physical Unclonable Function for Key Generation and Chip Identification.” In: *Information Security Journal: A Global Perspective* (2013).

[404] Mingshen Sun et al. “Blender: Self-randomizing Address Space Layout for Android Apps.” In: *RAID* (2016).

[405] Minh Tran et al. “On the Expressiveness of Return-into-libc Attacks.” In: *RAID* (2011).

[406] Muhammad Sabir Idrees et al. “Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates.” In: *Nets4Cars* (2011).

[407] Nadia Heninger et al. “Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.” In: *USENIX Security Symposium* (2012).

[408] Nathan Burow et al. “Control-Flow Integrity: Precision, Security, and Performance.” In: *CSUR* (2017).

[409] Nathan Cooprider et al. “Efficient Memory Safety for TinyOS.” In: *SenSys* (2007).

[410] Nicholas Carlini et al. “ROP is still dangerous: Breaking modern defenses.” In: *USENIX Security* (2014).

[411] Nick Christoulakis et al. “HCFI: Hardware-enforced Control-Flow Integrity.” In: *CODASPY* (2016).

[412] Nicolas Carlini et al. “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.” In: *USENIX Security* (2015).

[413] Nicolas Falliere et al. “W32.Stuxnet Dossier.” In: (2011). URL: https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[414] Niels Ferguson et al. *Practical Cryptography*. Wiley, 2003.

[415] Oliver Hahm et al. “Operating Systems for Low-End Devices in the Internet of Things: a Survey.” In: *IEEE Internet of Things Journal* (2016).

[416] Oscar Guillen et al. *Crypto-Bootloader – Secure in-field firmware updates for ultra-low power MCUs*. 2015.

[417] Oxana Andreeva et al. *Industrial Control Systems Vulnerabilities Statistics*. 2016. URL: https://kasperskycontenthub.com/securelist/files/2016/07/KL_REPORT_ICS_Statistic_vulnerabilities.pdf.

[418] Per Larsen et al. "SoK: Automated Software Diversity." In: *IEEE Security and Privacy* (2014).

[419] Peter Silberman et al. *A Comparison of Buffer Overflow Prevention Implementations and Weaknesses*. 2004. URL: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf.

[420] Philip Koopman et al. "Embedded System Design Issues (The Rest of the Story)." In: *Proceedings of the 1996 International Conference on Computer Design* (1996).

[421] Philip Koopman et al. "Embedded system security." In: *IEEE Computer* 7.37 (2004), 95—97.

[422] Philip Koopman et al. "Challenges in Deeply Networked System Survivability." In: *Proceedings of the NATO Advanced Research Workshop on Security and Embedded Systems* (2005).

[423] Philip Koopman et al. "Deeply Embedded Survivability." In: (2007).

[424] Priyanka Bagade et al. "Protect your BSN: No Handshakes, just Namaste!" In: *BSN* (2013).

[425] Ralf Spenneberg et al. *PLC-Blaster: A Worm Living Solely in the PLC*. 2016. URL: https://www.blackhat.com/docs/asia-16/materials/asia-16-Spenneberg-PLC-Blaster-A-Worm-Living-Solely-In-The-PLC-wp.pdf.

[426] Robert Hundt et al. "MAO – An extensible micro-architectural optimizer." In: *CGO* (2011).

[427] Robert J. Shaw et al. "Unified Forensic Methodology for the Analysis of Embedded Systems." In: *CACCT* (2010).

[428] Roberto Doriguzzi Corin et al. "TinyKey: A light-weight architecture for Wireless Sensor Networks securing real-world applications." In: *WONS* (2011).

[429] Roel Verdult et al. "Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer." In: *USENIX Security* (2013).

[430] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications." In: *TISSEC* (2012).

[431] Sabina Jeschke et al. *Industrial Internet of Things: Cybermanufacturing Systems*. Springer International Publishing, 2017.

[432] Sahar Mazloom et al. "A Security Analysis of an In Vehicle Infotainment and App Platform." In: *WOOT* (2016). URL: https://www.usenix.org/system/files/conference/woot16/woot16-paper-mazloom.pdf.

[433] Sandro Etalle et al. *Monitoring Industrial Control Systems to improve operations and security*. 2013.

[434] Sebastien Marineau-Mes et al. *The 27 Year Old Microkernel*. 2008. URL: http://community.qnx.com/sf/wiki/do/viewAttachment/projects.core_os/wiki/Oct27_Microkernel_Innovation/Webinar_kernel_oct07_final.ppt.

[435] Sergey Bratus et al. "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation." In: *USENIX ;login:* (2011).

[436] Sergey Bratus et al. "Composition Patterns of Hacking." In: *Cyberpatterns* (2012).

[437] Sergey Bratus et al. "'Weird Machine' Patterns." In: *Cyberpatterns* (2014).

[438] Sergey Shekyan et al. *To Watch Or To Be Watched. Turning your surveillance camera against you*. 2013. URL: https://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Sergey%20Shekyan%20and%20Artem%20Harutyunyan%20-%20Turning%20Your%20Surveillance%20Camera%20Against%20You.pdf.

[439] Sergio Pastrana et al. "AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices." In: *DIMVA* (2016).

[440] SeungJin Lee et al. *Hacking, Surveilling and Deceiving Victims on Smart TV*. 2013. URL: https://media.blackhat.com/us-13/US-13-Lee-Hacking-Surveilling-and-Deceiving-Victims-on-Smart-TV-Slides.pdf.

[441] Shohreh Hosseinzadeh et al. "Security in the Internet of Things through Obfuscation and Diversification." In: *ICCCS* (2015).

[442] Siddhartha Kumar Khaitan et al. "Design Techniques and Applications of Cyber Physical Systems: A Survey." In: *IEEE Systems Journal* (2015).

[443] Stefan Tillich et al. "Security Analysis of an Open Car Immobilizer Protocol Stack." In: *INTRUST* (2012).

[444] Stephen Checkoway et al. "Comprehensive Experimental Analyses of Automotive Attack Surfaces." In: *USENIX Security* (2011).

[445] Stephen Crane et al. "It's a TRAP: Table randomization and protection against function reuse attacks." In: *CCS* (2015).

[446] Stephen Crane et al. "Readactor: Practical code randomization resilient to memory disclosure." In: *IEEE Symposium on Security and Privacy* (2015).

[447] Stephen Crane et al. “Code Randomization: Haven’t We Solved This Problem Yet?” In: *SecDev* (2016).

[448] Stephen McLaughlin et al. “Embedded Firmware Diversity for Smart Electric Meters.” In: *HOTSEC* (2010). URL: https://www.usenix.org/legacy/event/hotsec10/tech/full_papers/McLaughlin.pdf.

[449] Stephen McLaughlin et al. “The Cybersecurity Landscape in Industrial Control Systems.” In: *Proceedings of the IEEE* (2016).

[450] Taeill Yoo et al. “Recoverable Random Numbers in an Internet of Things Operating System.” In: *Entropy* (2017).

[451] Tamara Bonaci et al. “To Make a Robot Secure: An Experimental Analysis of Cyber Security Threats Against Teleoperated Surgical Robots.” In: (2015). URL: http://brl.ee.washington.edu/wp-content/uploads/2014/05/arXiv_April_2015.pdf.

[452] Tavis Ormandy et al. *Linux ASLR Curiosities*. 2009. URL: https://www.cr0.org/paper/to-jt-linux-alsr-leak.pdf.

[453] Thomas Willhalm et al. *Intel Performance Counter Monitor - A better way to measure CPU utilization*. 2017. URL: http://www.intel.com/software/pcm.

[454] Tim Kornau et al. “Return oriented programming for the ARM architecture.” In: (2010). URL: https://static.googleusercontent.com/media/www.zynamics.com/nl//downloads/kornau-tim--diplomarbeit--rop.pdf.

[455] Todd Jackson et al. “Compiler-generated software diversity.” In: *Moving Target Defense, volume 54* (2011).

[456] Todd Jackson et al. “Diversifying the software stack using randomized NOP insertion.” In: *Moving Target Defense II, volume 100* (2013).

[457] Travis Goodspeed et al. “Half-Blind Attacks: Mask ROM Bootloaders are Dangerous.” In: *WOOT* (2009). URL: http://www.inrialpes.fr/planete/people/francill/Papers/Half_Blind_Attack_MSP430.pdf.

[458] Tyler Bletsch et al. “Code-reuse attacks: new frontiers and defenses.” In: (2011). URL: http://people.duke.edu/~tkb13/pubs/dissertation.pdf.

[459] Victor van der Veen et al. “Memory Errors: The Past, the Present, and the Future.” In: *RAID* (2012).

[460] Viktor Fischer et al. “True Random Number Generator Embedded in Reconfigurable Hardware.” In: *CHES* (2002).

[461] Viktor Fischer et al. “Random Number Generators for Cryptography: Design and Evaluation.” In: *Summer School on Design and Security of Cryptographic Algorithms and Devices* (2014).

[462] Wolfgang Puffitsch et al. “Time-Predictable Virtual Memory.” In: *ISORC* (2016).

[463] Xueyang Wang et al. “ConFirm: Detecting firmware modifications in embedded systems using Hardware Performance Counters.” In: *ICCAD* (2015).

[464] Yasser Shoukry et al. “Non-invasive Spoofing Attacks For Anti-lock Braking Systems.” In: *CHES* (2015).

[465] Yu Ding et al. “Android Low Entropy Demystified.” In: *ICC* (2014).

[466] Zach Lanier et al. *Voight-Kampff'ing The BlackBerry PlayBook*. 2012. URL: https://speakerdeck.com/quine/voight-kampffing-the-blackberry-playbook-v2.

[467] Zach Lanier et al. *No Apology Required: Deconstructing BB10*. 2014. URL: https://speakerdeck.com/duosec/no-apology-required-deconstructing-bb10.

[468] boredhackerblog. *How we broke into your house*. 2016. URL: http://www.boredhackerblog.info/2016/02/how-we-broke-into-your-house.html.

[469] Corelan Team (corelancod3r). *mona*. URL: https://github.com/corelan/mona.

[470] Corelan Team (corelancod3r). *Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR*. 2009. URL: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/.

[471] Corelan Team (corelancod3r). *Exploit writing tutorial part 10: Chaining DEP with ROP - the Rubik's[TM] Cube*. 2010. URL: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/.

[472] Corelan Team (corelancod3r). *Universal DEP/ASLR bypass with msvcr71.dll and mona.py*. 2011. URL: https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/.

[473] Clang 5 documentation. *Control Flow Integrity*. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[474] grsecurity. *Frequently Asked Questions About RAP*. URL: https://grsecurity.net/rap_faq.php.

[475] grsecurity. *grsecurity*. 2017. URL: https://www.grsecurity.net/.

[476] *http://www.rust-embedded.org/*. 2017. URL: http://intermezzos.github.io/.

[477] jmaxxz. *Backdooring the Frontdoor: Hacking a 'perfectly secure' smart lock*. 2016. URL: https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Jmaxxz-Backdooring-the-Frontdoor.pdf.

[478] Boris Škorić. *Physical aspects of digital security - Lecture Notes*. 2014. URL: http://security1.win.tue.nl/~bskoric/physsec/files/LectureNotes_2IMS10_v1.7.pdf.

[479] quintessenz. *Hacking CCTV: A Private Investigation*. 2005. URL: http://www.quintessenz.at/22c3/cctv_hacking_22c3.pdf.

[480] *zinc.rs*. 2017. URL: https://zinc.rs/.