PROSPECT

Peripheral Proxying Supported Embedded Code Testing

Markus Kammerstetter
Vienna University of Technology
mk@iseclab.org

Christian Platzer
Vienna University of Technology
cplatzer@iseclab.org

Wolfgang Kastner
Vienna University of Technology
k@auto.tuwien.ac.at

ABSTRACT

Embedded systems are an integral part of almost every electronic product today. From consumer electronics to industrial components in SCADA systems, their possible fields of application are manifold. While especially in industrial and critical infrastructures the security requirements are high, recent publications have shown that embedded systems do not cope well with this demand. One of the reasons is that embedded systems are being less scrutinized as embedded security analysis is considered to be more time consuming and challenging in comparison to PC systems. One of the key challenges on proprietary, resource constrained embedded devices is dynamic code analysis. The devices typically do not have the capabilities for a full-scale dynamic security evaluation. Likewise, the analyst cannot execute the software implementation inside a virtual machine due to the missing peripheral hardware that is required by the software to run. In this paper, we present PROSPECT, a system that can overcome these shortcomings and enables dynamic code analysis of embedded binary code inside arbitrary analysis environments. By transparently forwarding peripheral hardware accesses from the original host system into a virtual machine, PROSPECT allows security analysts to run the embedded software implementation without the need to know which and how embedded peripheral hardware components are accessed. We evaluated PROSPECT with respect to the performance impact and conducted a case study by doing a full-scale security audit of a widely used commercial fire alarm system in the building automation domain. Our results show that PROSPECT is both practical and usable for real-world application.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems; C.2.0 [**General**]: Security and protection

ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590301.

Keywords

embedded system, security, device tunneling, dynamic analysis, fuzz testing

1. INTRODUCTION

Embedded systems are omnipresent in today's world. From small digital clocks over home appliances such as washing machines and multimedia devices to medical appliances or smart phones, embedded technology provides tremendous advantages compared to general purpose systems. One key aspect is the possibility to create tailored hardware devices to fulfill a very specific task. With exactly the right amount of memory, processing power and interfaces, embedded devices are cheaper, smaller and faster than their general-purpose computing counterparts. However, a good amount of embedded devices are aimed at functionality rather than security. In fact, recent publications have shown that the security of embedded devices is especially bad [23, 8, 10, 14]. One reason is, that security audits on embedded devices are considered to be far more challenging and time consuming than on general purpose PC systems. Considering the common case in which the security analyst has no access to the source code of the system under test, there is a broad gap between state-of-the-art security analysis techniques for PCs and for embedded systems. Liu et al. [11] and Austin et al. [1] give an overview of the wide area of vulnerability discovery techniques that are available for PC systems. The techniques range from sophisticated static analysis techniques over dynamic analysis and fuzz testing to advanced dynamic taint analysis and symbolic or concolic execution [16, 4]. However, for embedded systems, the situation is different. Mainly due to custom proprietary hardware, undocumented peripherals and strict system limitations, the prevalent vulnerability discovery techniques are still based on static analysis [9, 3, 22].

At the same time, using the wide range of dynamic analysis or taint analysis and symbolic execution tools is in general not possible due to the limitations of the embedded system under test. One solution would be to take the investigated application from its original context and run it in a virtual machine that provides the necessary resources and facilities for a full dynamic evaluation. To emulate the embedded device, however, the connected peripheral hardware needs to be available from within the virtual machine as well. The usual way is to emulate the peripheral hardware. Yet, for proprietary hardware this is not possible due to the following reasons. First, the analyst would need comprehensive information on how all peripheral hardware devices work in order

to emulate the hardware behavior in software. Since peripheral hardware is likely to be proprietary, this information is not available and, subsequently, the analyst can not emulate the hardware. Second, even if the information is available to the analyst, adding full support for new peripheral hardware components to a virtual machine implementation is not an easy task. It is likely that the implementation would take the analyst a tremendous amount of time that renders the whole dynamic security analysis infeasible.

To amend this problem, we take a different approach and introduce PROSPECT, a proxy capable of tunneling arbitrary peripheral hardware accesses from within a virtual machine to the embedded system under test. The result is a virtualized execution environment for embedded software implementations with a completely transparent connection to the actual peripheral hardware components of the system under test. PROSPECT thus enables the analyst to leverage any powerful dynamic analysis techniques of her choice to discover vulnerabilities on embedded devices with minimal effort. We developed and continuously improved PROSPECT over a duration of more than 10 months during which our system evolved. In addition, we conducted a case study to prove the effectiveness of PROSPECT and used the system to undertake a full scale security analysis of a widely used proprietary fire alarm system in the building automation domain. Summing up, the contributions presented in this paper are as follows:

- We introduce PROSPECT, a transparent proxy for tunneling peripheral hardware accesses from within a virtual analysis environment to the embedded system under test. Our system can overcome prevalent analysis limitations by enabling dynamic instrumentation inside arbitrary analysis environments.
- We provide a MIPS based proof-of-concept implementation that has continuously evolved over a duration of more than 10 months.
- We evaluate and discuss our approach with a detailed analysis of the system's performance and usability.
- We utilized PROSPECT to conduct a case study by running a full-scale security audit of a widely used commercial fire alarm system in the building automation domain showing that PROSPECT is both practical and usable for real-world application.

2. CHALLENGES IN EMBEDDED SECURITY ANALYSIS

Assuming that the reader is familiar with the general field of information security, in this chapter, we briefly outline binary code analysis and highlight fuzz testing as exemplary, widely established techniques to discover software vulnerabilities. We point out, that dynamic analysis is one of the key requirements for efficient fuzz testing as well as for manual in-depth analysis approaches usually done as soon as fuzz testing discovers a potential security vulnerability. After presenting dynamic analysis techniques for PC systems, we continue by providing a general overview of how typical medium to large scale embedded systems are made up and why the presented dynamic analysis approaches are frequently not applicable to embedded systems. Besides, the chapter shows why the approach PROSPECT takes is promising as it can overcome the described challenges and enable dynamic analysis in general, regardless of the analysis limitations on the system under test.

2.1 Binary Code Analysis

Vendors are typically profit-driven and try to push their newest software products to market as soon as possible. Depending on their efforts to avoid software vulnerabilities, a released software implementation may contain numerous security flaws such as stack smashing or use-after-free vulnerabilities [17]. At that point, an arms race between attackers and the vendor begins. Attackers try to exploit the vulnerabilities for their own ill-gotten gain such as industrial espionage, spreading malware or setting up botnets [20, 7] while vendors try to patch newly discovered bugs.

For proprietary software implementations, the source code is usually not available. Thus, in order to discover vulnerabilities in these implementations, security analysts need to rely on techniques that can be applied to binary code. In a recent survey [11], Liu et al. describe a number of common techniques to discover software vulnerabilities. While analysts can resort to static analysis that does not require the execution of the program under test, static analysis suffers from a number of drawbacks hindering penetration tests. For instance, object orientated code makes frequent use of function pointers that are hard to resolve, if the program is not being executed. With dynamic analysis, the program is being executed and the analyst can trace and instrument the current execution path of the program under test. However, unless advanced techniques such as multipath exploration [13] are employed, the analyst needs to generate different program inputs to analyze different execution paths.

2.2 Fuzz Testing: A Common Technique to discover Software Vulnerabilities

Generating different program inputs to reach different execution paths is also one of the key ideas of *fuzz testing*, a widely established technique to discover software vulnerabilities [11, 5, 2]. With fuzz testing, input data to the program are generated automatically either at random or by mutating previously obtained program input. At the same time, the analyst can employ dynamic analysis to monitor the program execution and detect program anomalies such as crashes, illicit memory accesses or endless loops causing high CPU utilization. If an anomaly is detected, the generated input data are likely to have caused the abnormal behavior. This is a starting point for a more thorough manual program analysis, usually also within a dynamic analysis environment.

Practical results [11, 17, 5, 2] have shown that fuzzing is both a viable and established technique to discover software vulnerabilities. However, since fuzzers can be highly application specific, it might be necessary to implement new fuzzing tools for each penetration test. Also, we would like to stress that although fuzz testing is widely used, it is not the only technique to discovery software vulnerabilities efficiently. One key observation at this point is that dynamic instrumentation is required for both efficient fuzz testing and the manual analysis that is usually done after the fuzzer discovers a potential security vulnerability.

2.3 Dynamic Code Analysis on PC Systems

In general, a PC system can be divided into hardware, an operating system (including kernel and drivers) and software applications. The analyst has the freedom to dynamically instrument any of these layers. The easiest way to instrument a program is to debug it with a state-of-the-art-debugger such as *gdb* or *Ida Pro*. The drawback here is that the program can easily detect that it is being instrumented and behave differently. For instance, the program might just exit instead of performing its usual functionality. On the next level, the analyst can instrument the operating system to analyze the program's behavior. For instance, CWSandbox [24] uses this approach by hooking the operating system libraries. This allows the analyst to trace the behavior of the application, but at the same time hinders typical techniques for debugging (e.g. single stepping through code). On the lowest level, the analyst can instrument the hardware using Virtual Machine Introspection (VMI) [6], which makes it hard for the investigated software to detect that it is being analyzed. Although the target of the analysis is the application itself, the downside of this approach is the need to analyze the surrounding operating system as well. Therefore, the necessary effort is higher than applying a regular debugging technique.

2.4 A typical Embedded System

Embedded systems can be divided into small, medium and large scale embedded systems [19, 14]. Depending on their size, their system configuration can differ tremendously. Small scale embedded systems such as electronic toys, digital clocks or pocket calculators are built around strongly resource constrained microcontrollers. Typically, there is no operating system and the firmware of these systems comprises a single program that is contained in an on-chip Flash memory. In contrast, medium and large scale embedded systems such as smart phones, wireless routers or field level components of SCADA systems are based on more powerful controllers. Typically, they run a customized operating system (e.g. Linux) and the product-specific implementation of an embedded product often comprises custom kernel code, drivers and several applications. At the heart of these systems commonly lies a powerful System-On-Chip (SoC) controller that includes a CPU, ROM, SRAM and a number of internal peripherals and I/O controllers.

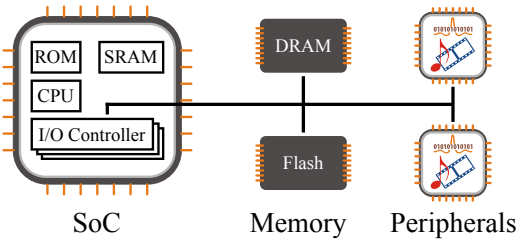

Figure 1: A Typical Medium to Large Scale Embedded System

A typical separation of components is shown in Figure 1. Upon power-up, the CPU in the SoC controller will execute the first-stage bootloader code contained in internal ROM and perform low-level initializations. After that, the SoC can access external memories (such as Flash and SDRAM) to boot into a second-stage bootloader and, consequently, into the operating system (OS) kernel. At that point, the OS can load a number of additional drivers to support external peripherals and then start the product specific processes. While the general operation of embedded systems is similar to PC systems, it is the external peripherals that make embedded systems so special. External peripherals are typically customly designed by the system manufacturers ranging from product specific sensors and actuators to custom communication interfaces. Taking modern smart phones as an example, such external peripherals could be charging controls, wireless radios, GPS receivers, magnetic or accelerations sensors, driving circuits for the vibrating alert, speech compression DSPs and many more. These external peripherals are what actually transforms an off-the-shelf SoC system into a valuable everyday product.

2.5 Challenges of Dynamic Code Analysis on Embedded Systems

In contrast to PC systems, employing dynamic analysis techniques on embedded systems can be more challenging. Typically, embedded devices are resource constrained, access to the file system is limited and the kernel's functionality and tools available on the device are just a minimal set of functions necessary for the product to operate properly [14]. The main reason for these constraints is that including additional functionality on the embedded systems would result in increased embedded resource requirements and ultimately in higher manufacturing costs. From this perspective, the presented analysis approaches for PC systems are hard to apply to their embedded counterparts:

1. Using a debugger to instrument the program is only feasible if the OS kernel includes debugging support (e.g. through `ptrace()` in the case of Linux). Running a state-of-the-art debugger on the system might not be possible due to resource constraints (e.g. in terms of memory consumption) or due to missing support (e.g on legacy systems or on systems where `ptrace()` support was not compiled into the kernel to save memory space).

2. Instrumenting the operating system would require kernel modifications or loading custom kernel modules. Embedded systems often run customized minimal kernel configurations to keep resource consumption and boot-up delays low. As a result, instrumenting the operating system might not be feasible.

3. Instrumenting the hardware would require not only virtualization of the system architecture, but also of all the necessary peripheral devices. However, as peripheral devices and their drivers are often proprietary, the information required to emulate them might not be publicly available. Besides, writing emulation code for all peripheral hardware devices would cause a tremendous overhead, considering that the analyst's goal is dynamic code analysis of only a small set of programs.

These challenges show that while on PC systems there is a wide range of established and well working vulnerability discovery techniques, the situation is different on embedded systems. In theory, all of those techniques could be applied to embedded systems as well. However, practically, embedded systems frequently lack support for these techniques and

thus make it much harder to discover software vulnerabilities. We believe that this is also the reason, why static analysis techniques are still so prevalent for those systems.

3. PERIPHERAL DEVICE FORWARDING

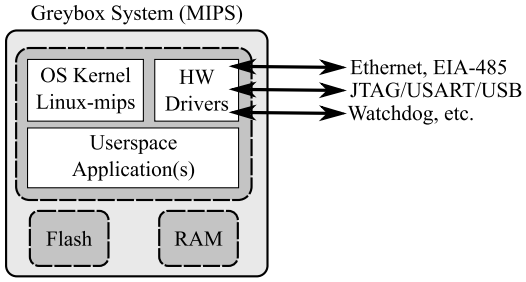

Figure 2: A typical Greybox System Example

Figure 2 shows a typical greybox embedded system example from a security analyst's point of view. The analyst's goal is to test one or more userspace applications on the embedded system for security vulnerabilities. This could, for instance, be a network daemon that is exposed to external attackers over a network connection. Yet, due to the challenges portrayed in Section 2, the analyst is unable to perform dynamic code analysis on the target system.

That is, the system lacks system resources, analysis tools are not available or can not be run and the userspace application, the analyst is interested in, can not be executed in a virtual environment as the peripheral hardware is missing there. However, one key observation is that userspace applications commonly communicate through character devices with potentially proprietary drivers and, consequently, with the peripheral hardware. Also, the communication interfaces to exchange data with the driver, and therefore with the kernel, are limited and can be considered standardized.

This is where PROSPECT comes into play. The basic idea of the system is to create virtual character devices inside another physical or virtual analysis environment. PROSPECT must intercept system calls used for communication with the character device from within the operating system kernel, forward them to the appropriate device on the embedded system and execute them there. Any responses need to be fed back to the analysis environment so that the intercepted system calls can return the data from the embedded remote system. Block devices, on the other hand, are generally used to access storage media which are emulated by the analysis virtual machine (i.e. qemu) anyways. To the analyst, PROSPECT constitutes a transparent forwarding solution for character device communication and thus allows her to conduct dynamic analysis techniques that were previously infeasible. Even software running on legacy systems lacking support for state-of-the-art analysis tools can be analyzed this way. As a result, PROSPECT allows to overcome typical challenges an embedded security analyst typically needs to face today.

3.1 Character Device Access

In order to forward peripheral hardware accesses, we need to know which system calls are generally used to interact with character devices. Targeting Linux systems, we gathered information on the supported `file_operations` of all included character device drivers in three different Linux kernel versions (Linux-2.4.20, Linux-2.6.38.1 and Linux-3.4.4) by analyzing the source code of all available drivers (514 files in total). We chose these specific kernel versions to get an idea which system calls are used to access character devices on legacy systems (i.e. Linux-2.4 and Linux-2.6) as well as on current kernel versions (i.e. Linux-3.4). Table 1 shows how many of the character device driver source code files actually define `file_operations`. It can be seen that the number of files decreases with newer kernels. We believe that this is due to increased abstraction in the Linux kernel requiring driver authors to write less supporting code.

Linux-2.4.20			Linux-2.6.38.1			Linux-3.4.4		
files	fops	fops %	files	fops	fops %	files	fops	fops %
264	77	29.17	143	62	43.36	107	54	50.47

Table 1: Analyzed Device Drivers on different Linux Kernel Versions

Table 2 shows which `file_operations` (i.e. which system calls) are used to interact with character device drivers in the different Linux kernel versions in relation to the number character device source code files. For instance, on Linux-2.4.20, there are 77 files that define `file_operations` and, out of these, 83.12% define a custom handler for the `open` system call. Some of the system calls in older kernel versions have been superseded by newer ones. For instance, the `ioctl` call in Linux-2.4.20 has been replaced by `unlocked_ioctl` and `compat_ioctl` for performance reasons in newer kernel versions.

Syscall	2.4.20	2.6.38.1	3.4.4
aio_fsync	-	0.00	0.00
aio_read	-	1.61	1.85
aio_write	-	1.61	1.85
check_flags	-	0.00	0.00
compat_ioctl	-	6.45	7.41
fallocate	-	0.00	0.00
fasync	28.57	11.29	12.96
flock	-	0.00	0.00
flush	14.29	-	-
fsync	0.00	3.23	3.70
get_unmapped_area	2.60	1.61	1.85
ioctl	84.42	-	-
llseek	6.49	32.26	29.63
lock	0.00	0.00	0.00
mmap	18.18	12.90	14.81
open	83.12	74.19	77.78
poll	32.47	20.97	25.93
read	68.83	82.26	85.19
readdir	0.00	0.00	0.00
readv	0.00	-	-
release	77.92	62.90	66.67
sendpage	0.00	0.00	0.00
setlease	-	0.00	0.00
splice_read	-	0.00	0.00
splice_write	-	0.00	0.00
unlocked_ioctl	-	51.61	50.00
write	62.34	50.00	55.56
writev	0.00	-	-

Table 2: Usage of Linux `file_operations` in Character Device Drivers (Percentage) for different Linux versions

In theory, PROSPECT could forward any of the system calls visible in Table 2. However, for performance reasons, it is beneficial to handle some of those calls locally. In fact, PROSPECT could execute system calls such as `flush`, `sync`, `fasync` or `aio_fsync` locally, if device access on the remote device is kept synchronized. Due to the delay imposed by the connection to the remote system, however, this would have a negligible effect. On the other side, system calls such as `splice_read`, that have been introduced for performance reasons, can use their regular counterparts (i.e. `read`) without breaking their basic functionality. With PROSPECT, we thus focus on *basic* character device operations that are broadly used by the majority of the character device drivers we analyzed. Specifically, PROSPECT can handle the `file_operations` listed in Table 3. The first column shows the name of the system call. For each system call, we specify if the system call is supported by PROSPECT, which system call is used to implement it and whether the call is handled locally or forwarded to the remote system.

Syscall	supported	implemented through	local/ remote
aio_fsync	yes	fsync	local
aio_read	yes	read	remote
aio_write	yes	write	remote
check_flags	no	-	-
compat_ioctl	yes	ioctl	remote
fallocate	no	-	-
fasync	yes	fsync	local
flock	no	-	-
flush	yes	fsync	local
fsync	yes	fsync	local
get_unmapped_area	no	-	-
ioctl	yes	ioctl	-
llseek	yes	llseek	remote
lock	no	-	-
mmap	no	-	-
open	yes	open	remote
poll	yes	poll	remote
read	yes	read	remote
readdir	no	-	-
readv	no	-	-
release	yes	release	remote
sendpage	no	-	-
setlease	no	-	-
splice_read	yes	read	remote
splice_write	yes	write	remote
unlocked_ioctl	yes	ioctl	remote
write	yes	write	remote
writev	no	-	-

Table 3: Basic `file_operations` supported by PROSPECT

With the exception of `mmap`, all operations that are frequently used for character device communication are supported. This is due to the fact that FUSE (Filesystem in Userspace) does not support direct `mmap` calls for character devices at the moment (see Section 7).

However, special consideration was necessary for the `ioctl` system call, as its exact behavior can frequently not be inferred prior to the actual execution.

3.2 IOCTL Mechanism and its Shortcomings

The IOCTL (I/O control) mechanism allows more flexibility in the communication with underlying device files. In general, there are two types of IOCTLs: Well-formed and unrestrictive IOCTLs. The difference is that well-formed IOCTLs have information about the type of the call encoded in the request number, whereas unrestricted IOCTLs do not provide this kind of information.

```
#define _IOC(dir,type,nr,size)    \
((dir  << _IOC_DIRSHIFT)|(type << _IOC_TYPESHIFT)|\
 (nr   << _IOC_NRSHIFT) |(size << _IOC_SIZESHIFT))
```

Listing 1: Encoding for well-formed IOCTLs

Listing 1 shows how information such as the direction (e.g. read, write or none/execute), the request number or the amount of data (size) are encoded in the `ioctl` request parameter. By decoding this parameter prior to the actual `ioctl` call, it is possible to determine the direction of the data transfer and whether the provided parameter is a constant or a pointer to memory. In this case, PROSPECT could forward the request to the target system.

However, for unrestricted IOCTLs and prior to the actual `ioctl` execution, PROSPECT would have no way to determine the direction of the data, how much data should be transferred and whether a provided parameter is supposed to be a pointer or not. Since unrestricted IOCTLs are commonly used for device driver communication, we had to address this issue in the design of PROSPECT. We solved the challenge by introducing a concept we denote *Dynamic Memory Tunneling* which is described in Section 4.3.

4. IMPLEMENTATION

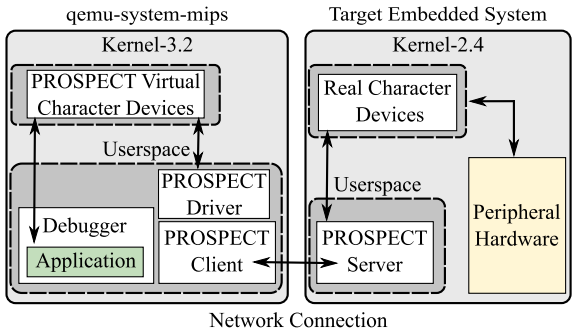

Figure 3: Peripheral Character Device Forwarding

As sketched in Sections 2.2 and 2.5, we assume that the analyst would like to dynamically instrument a binary on an embedded system that heavily accesses peripheral devices. While infeasible without PROSPECT, the application can now be executed inside an arbitrary analysis environment. Figure 3 provides a schematic overview of PROSPECT. On the left side, the application that should be analyzed is being executed within an arbitrary state-of-the-art debugger. However, instead of directly accessing the peripheral hardware through a character device, the application actually interacts with the *virtual* character devices that were generated through PROSPECT. At this point, PROSPECT intercepts the system calls defined in `file_operations` and forwards them to the userspace PROSPECT client. The client decides whether the system call should be executed locally or on the remote system. If remote execution is required, it communicates with the lightweight PROSPECT

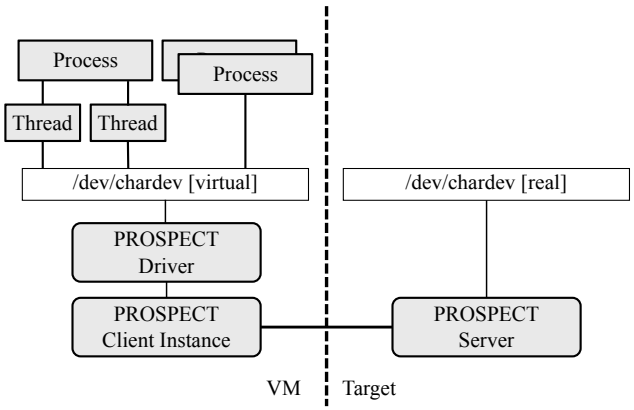

Figure 4: Concurrent Device Access

server on the target system. The server has very low system requirements and can thus be executed on a wide range of embedded systems. Once the system call has been executed on the target system, any results are fed back into the software application under analysis.

In order to generate virtual character devices and intercept system calls, parts of PROSPECT need to run in kernel context. While we could have realized all parts of PROSPECT in kernel space, we decided that the major part of our implementation should be in user space. In comparison to a full kernel space implementation, a user space centric implementation has the advantage of increased system stability, security and, most of all, more flexibility. We implemented PROSPECT from scratch and our overall implementation consists of roughly $7,500$ lines of C code. In summary, PROSPECT comprises:

- A lightweight kernel driver utilizing the FUSE [15] kernel framework.
- A userspace driver combined with the PROSPECT client
- A lightweight server component running on the target system.

The PROSPECT lightweight server needs to be executed on the target system. Assuming that the security analyst typically has full physical access to the embedded device under test, we believe that this is a viable option. For instance, the analyst could use the bootloader console to get root level access to the operating system and then simply copy the PROSPECT statically linked binary to the device by using an attached storage medium or a networked remote machine as source.

Since our lightweight kernel driver utilizes the FUSE kernel framework, PROSPECT has the advantage that it is applicable to a wide range of operating systems, including Linux, FreeBSD, NetBSD, OpenSolaris, Android and OS X.

4.1 Concurrent Device Accesses

On typical embedded systems, a character device might be accessed by multiple threads or processes concurrently. Likewise, a single process or thread might interact with multiple devices at the same time. PROSPECT can handle these scenarios by using a client/server architecture (Figure 3) with multiple synchronization mechanisms. On the target system, there is a single PROSPECT server that can handle multiple incoming connections. Each client represents a character device that is forwarded to the target system, whereas each client can handle concurrent devices accesses by multiple threads and/or processes (Figure 4). PROSPECT relies on POSIX thread synchronization mechanisms (i.e. mutexes) to sustain the order of all accesses throughout the system.

4.2 File Descriptor Tracking

PROSPECT needs to keep track of file descriptors. Essentially, there are three cases we need to consider: (1) single processes, (2) child processes (i.e. created with `fork()`) and (3) threads. As file descriptors work on a per-process basis, they are only unique within the context of a process. Whenever a new process is spawned, new file descriptors returned by `open()` typically start at 3 (as 0,1 and 2 are already used for *stdin*, *stdout* and *stderr*, respectively). Considering two different processes, both processes may receive the same file descriptor (i.e. 3), but it may correspond to completely different files with different properties (e.g. file offsets or permissions). If a process uses `fork()` to spawn a child process, it will inherit all open file descriptors from its parent, but any new file descriptors it receives at a later point will be unique to the child process. In contrast, threads behave much like a single process, as all file descriptors they receive are shared between them. As a result, both the PROSPECT client as well as the server would need to be aware of the type of process or thread in order to emulate normal operating system behavior.

PROSPECT tackles this challenge by taking a different approach. Instead of emulating the behavior of a real system, it uses *globally unique* file descriptors on the server side and supplies them in a synchronized way to all clients. More specifically, we implemented the PROSPECT server as a single process but with multiple threads to handle different connections. For that reason, all file descriptors it receives from the target's OS kernel are unique within the server and, ultimately, within all PROSPECT clients and all processes and/or threads accessing virtual character devices as well.

4.3 Dynamic Memory Tunneling

In Section 3.2, we explained how the IOCTL mechanism is used for more flexible device driver communication. However, unlike well-formed IOCTLs, their unrestricted counterparts do not provide any data exchange information (i.e. information on direction and the amount of data that should be exchanged with the driver). As unrestricted IOCTLs are frequently used, we considered different approaches to address this challenge in PROSPECT.

Both, the userspace application(s) accessing a character device as well as the character device driver are aware of the parameters for unrestricted IOCTLs. A userspace application may not use all unrestricted IOCTLs the driver supports. However, the driver implementation always includes all supported unrestricted IOCTLs as well as the information on how data can be exchanged with them. Even better, device driver code is typically structured in a known way so that it can be loaded by the operating system. For that reason, we could extract the kernel image and device drivers from the target system and employ static (or even dynamic)

code analysis techniques on the binaries to extract a data exchange rule-set for all available unrestricted IOCTLs. The drawback of this approach is that PROSPECT would need to be aware of operating system specifics (such as architecture, kernel version, kernel configuration, etc.). Thus, it would be hard to use PROSPECT on a wide range of different systems without major modifications. On the other side, extracting a rule-set from the userspace application, the analyst wants to work with, might be a challenge on its own (i.e. due to code size, program obfuscation or required manual code analysis).

Another approach we considered is that instead of extracting a rule set, PROSPECT could dynamically observe any unrestricted IOCTLs during program execution and learn from them. However, this is not always feasible, as the analysis would need to take place on the target system that does not necessarily support dynamic analysis in the first place. In fact, one of the goals of PROSPECT is to enable dynamic analysis on embedded systems, that might not support it for the reasons mentioned in Section 2.5.

Since any analysis required to gain information on unrestricted IOCTL parameters should not depend on the capabilities of the target system, we implemented *dynamic memory tunneling*. The key idea of dynamic memory tunneling is to always transfer a memory buffer to the target system if the IOCTL parameter is a possible pointer to a memory location. Accordingly, for each unrestricted IOCTL call, we need to answer the following questions:

- Is the parameter a valid pointer?
- How much data should be transferred to/from the target?

To determine whether the IOCTL parameter is a valid pointer, we use a heuristic. For each unrestricted IOCTL call, our system retrieves the PID (process ID) of the program that currently accesses the character device. For that PID it retrieves all mapped memory regions from the OS kernel (i.e. through `/proc/PID/maps`) and filters out any regions that are not suitable for a buffer (i.e. memory regions that are not read- and writable at the same time). If the parameter value is in one of the remaining memory regions, PROSPECT assumes that the parameter is a pointer and a data transfer with the target is initiated.

The question remains *how much* memory should be transferred to and from the target system. During our experiments we observed that the amount of data exchanged with unrestricted IOCTLs was below the page size (typically 4096 bytes on Linux) in all cases. To allow exceptions with larger buffer sizes, we experimentally limited the maximum size to 3∗PAGESIZE = 12KiB. However, PROSPECT can be easily reconfigured with increased limits. Besides the configured limit, the amount of memory that is actually transferred, can be limited through the mapped memory region boundaries as well.

During execution, for any unrestricted IOCTL call with a valid pointer as parameter, PROSPECT takes the following steps:

1. Given the pointer address ADDR and the PID of the calling process, use the kernel driver to read up to 3∗PAGESIZE bytes from the mapped memory region of the corresponding userspace process.
2. Transfer the buffer to the PROSPECT server on the target system and execute the unrestricted IOCTL call on a local copy of the transferred buffer.
3. Once `ioctl()` returns, compare the transferred buffer with the potentially modified local copy of the buffer to determine how many bytes were changed in the local copy.
4. In addition to the `ioctl()` *return* and *errno* values, send back the content of the local copy buffer to the corresponding client. The size of the transfer is limited through the last byte in the buffer that has actually changed (see Step 3).
5. Given the pointer address ADDR and the PID of the calling process, use the kernel driver to overwrite the corresponding memory region of the userspace process (i.e. starting at ADDR) with the content of the response buffer.
6. Return the `ioctl()` *return* and *errno* values to the calling process.

Through dynamic memory tunneling, PROSPECT can forward unrestricted IOCTLs with arbitrary read, write and execute operations.

5. EVALUATION

To provide a well-founded discussion of our system, we evaluated PROSPECT in two ways. First, we collected system call timing information to determine the performance impact PROSPECT causes in comparison with the native system. Second, we conducted a case study over more than 6 months by running a full-scale security audit of a widely used commercial fire alarm system in the building automation domain.

5.1 Evaluation of Performance Impact

On a 324 MHz embedded Linux MIPS system with 16MiB RAM, we used the *strace* tool to collect timing information for the system calls that are used for basic character device access (see Table 3 in Section 3.1 for details). Table 4 shows the userspace system calls we monitored.

Operation	Function
`close()`	Close device
`ioctl()`	I/O Control mechanism
`lseek()`	Seek to a given position
`_newselect()`	System call used for `poll()`
`open()`	Open device
`read()`	Read data from device
`write()`	Write data to device

Table 4: System Calls used for Character Device Access

To collect timing information, we ran a userspace application that makes heavy use of all of the system calls in Table 4. In order to determine how much longer the forwarded system calls take, we ran the application with PROSPECT in our analysis environment (qemu-system-mips) as well as natively on the embedded MIPS system. For both executions we used *strace* to create system call logs with timing

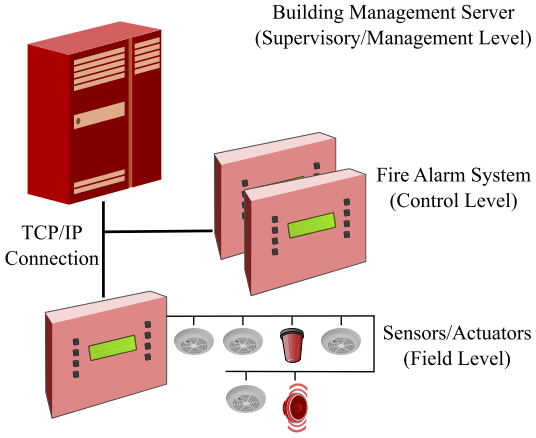

Figure 5: Proprietary Fire Alarm System

information. For our measurements we collected timing information for 196,075 system calls on the analysis environment and for 166,972 system calls on the embedded MIPS system. To compare the execution time of the system calls, we created custom analysis scripts to keep track of the file descriptors. This way, we were able to consider only timing information for calls made on character devices that are forwarded when PROSPECT is used. The results are visible in Table 5 in Section 6.1.

5.2 Case Study: Security Audit of a Proprietary Fire Alarm System

Under a legally binding non-disclosure agreement, we were able to employ PROSPECT to conduct a full-scale security audit of a widely used fire alarm system over a time frame of more than 6 months. A schematic overview of the overall fire alarm system is visible in Figure 5. On the lower side of the picture there is the fire alarm system that has a field level bus with a number of sensors (such as smoke detectors) and actuators (such as alarm lights or sirens) attached to it. Typically, there is one fire alarm system in a building and the sensors/actuators are situated in the rooms or on the outside of each building. Each fire alarm system is connected over a network connection (i.e. via TCP/IP) to one central building management server that is responsible for all fire alarm systems in multiple buildings. Thus, the server can manage the fire alarm systems and necessary steps can be taken in case there is a fire alarm. From the security perspective, the TCP/IP connection between the fire alarm systems and the building management server is interesting. After all, if an attacker could get access to the fire alarm systems over the building network or even the Internet, it might be possible to trigger false alarms or to disable fire alarms which would lead to a dangerous situation for the persons in the building.

On a technical level, the fire alarm system we analyzed is a customized embedded Linux system with custom drivers, custom peripheral hardware components and several proprietary userspace programs that make up the overall fire alarm system implementation. The userspace programs make heavy use of multi-threading (pthreads) and the `fork` mechanism. In the running state, there is a total of 29 multi-threaded fire-alarm system specific processes, spawning multiple threads depending on the handled networking communication. In total, there are 5 different hardware peripherals that are accessed concurrently by the different processes and threads. As soon as the whole system is up and running, any network communication is processed. The system resources are very limited and the fire alarm implementation consumes nearly all available resources.

Due to the resource constraints, it is not possible to run a debugger on the system. Thus, dynamic analysis on the device is not possible either and the code that handles the network communication cannot be analyzed in another environment, as the device specific peripheral hardware would be missing there. As a result, the software application(s) would not start up in the first place. In this case, the analyst would be limited to static analysis and/or very basic security testing techniques.

To conduct a security audit of the fire alarm system implementation that handles the network communication, we employed PROSPECT to run the fire alarm system software implementation inside a virtual analysis environment. In this case, we utilized *qemu*, an open source virtualization environment that also supports the MIPS architecture. The center of Figure 6 shows the multi threaded userspace application with all connected installations for a complete analysis. It concurrently interacts with 5 different peripheral devices which are handled by multiple PROSPECT client instances, each handling exactly one character device. For automated fuzz testing of the network protocol implementation, we set up three different machines. On the left, there is the Fire Alarm Control VM that runs the manufacturer's software to communicate with the fire alarm system over a network connection. We used this machine to generate network traffic and capture it (*Packet Capture*) to obtain packets that can be used as input data for our fuzzer. Accordingly, the fuzzer can generate randomized traffic that looks very similar to the original communication protocol by taking packets from the captured network traffic, randomizing a single byte at a random position within the packet each test run and replaying the communication towards the userspace application under test. This allows us to use a single fuzzer implementation for a broad range of proprietary network protocols without the need to know protocol specifics or the requirement to develop a new fuzzer for each protocol. The downside of this approach is the limitation of the test cases to the captured network communication: If feasible protocol states are not captured during the capture phase, our fuzzer will not be able to test them. At the same time, we used a debug server to run the userspace applications we want to analyze. Through the Debugger VM (with a state-of-the-art debugger), the fuzzer can thus monitor the state of the software application and whether the test packets it sent, caused an exception such as a memory access violation. In this case, the fuzzer stores the network packets that led to the exception for later (manual) analysis.

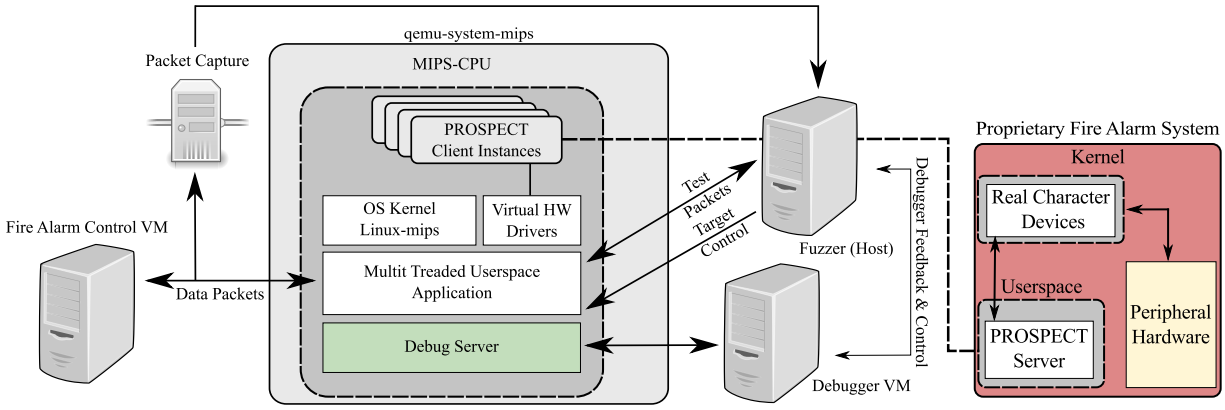

Figure 6: Security Analysis Environment

6. RESULTS AND DISCUSSION

6.1 Performance Impact

Table 5 shows the average performance impact of our system (Section 5.1). The values are arithmetic means over all recorded system calls. More specifically, the average number of `read()` accesses is the average of all read accesses from 166,972 native and 196,075 forwarded system calls, respectively. It clearly shows that for system calls that can be forwarded without further consideration (i.e. `lseek()`, `read()`, `write()` and `_newselect()`), the slowdown is practically insignificant as the main use of PROSPECT is program debugging (i.e. single stepping) and dynamic code analysis. Our results show that with PROSPECT the `_newselect()` call was slightly faster than on the native system. This is due to the nature of the system call. It blocks as long as either the given timeout is reached or one of the monitored file descriptors is ready. As this is closely related to the behavior of peripheral hardware (e.g. sleep modes), small variances in the recorded values are unavoidable.

In contrast, the `ioctl()` and `open()` calls cause a significant slowdown. The `ioctl()` slowdown is caused by the dynamic memory tunneling mechanism described in Section 4.3 whereas the `open()` slowdown is due to the connection establishment between the PROSPECT client and server. On the virtual analysis environment, we were unable to capture `close()` calls on virtual character devices which is why we can not provide a performance comparison. However, as `close()` also works on an existing PROSPECT connection and no special considerations are necessary for the call, we believe that the performance impact is comparable to the `lseek()`, `read()`, `write()` and `_newselect()` calls. Furthermore, Table 5 also lists the frequency of each specific system call. It shows that the most frequently used calls are also the fastest. For instance, `write()`, `read()`, `seek()` and `_newselect()` account for 95,71% of all forwarded system calls. The distribution between system calls for native and forwarded execution slightly varied between analysis runs due to the internal state of the peripheral hardware.

6.2 Proprietary Fire Alarm System Security Audit

During our fire alarm security analysis (Section 5.2), we conducted extensive fuzz testing with the setup shown in Figure 6. During analysis, PROSPECT successfully forwarded more than 500,000 system calls per analysis run to the target system without issues. Likewise, we were able to manually debug and single-step through the fire alarm application code. Our fuzz tests revealed a previously unknown zero-day vulnerability that was reported to the manufacturer. Our case study shows that even under demanding real-life requirements (29 multi-threaded processes that concurrently access 5 different hardware peripherals), our system performed well and enabled us to conduct both dynamic analysis and extensive fuzz testing to discover vulnerabilities.

7. LIMITATIONS AND FUTURE WORK

Due to the nature of PROSPECT, it has a number of limitations that need to be considered. At the moment, our system uses TCP/IP over a network connection between the virtual analysis system and the embedded target system. However, if the userspace application under analysis changes the network configuration, this would also bring down the PROSPECT connection. Similarly, if the target system has no network interface, PROSPECT can not be used. We plan to add support for different communication interfaces (such as serial links) to PROSPECT so that it is usable in these cases as well. Another limitation is that PROSPECT requires *pthreads* on the target system and only runs on Linux right now. Another limitation is the missing `mmap` support for character devices due to the missing support in FUSE. Since `mmap` calls can be forwarded just the same (i.e. by using a similar approach as described in Section 4.3), we plan to implement full `mmap` support in future versions.

As PROSPECT requires only very little supported functionality on the target system and FUSE has been ported to a number of operating systems (Section 3), our system could be easily ported to different architectures and operating systems as well. At least for some implementations, the considerable slowdown caused by PROSPECT might lead to issues. However, this situation also occurs when single-stepping through programs and solutions, such as altering the information returned by timing related system calls, exist. Also, our system does not provide any security features at the moment.

Another consideration was briefly discussed in Section 3. When accessing devices on UNIX systems, their access rights are determined by the device's permissions. The client implementation needs to create virtual character devices and

Syscall	Native [%]	Native [ms]	Fwd. [%]	Fwd. [ms]	Diff. [ms]	Slowdown [x]
`write()`	21.27	6.07	22.79	25.39	19.32	3.18
`lseek()`	28.14	0.12	18.88	2.31	2.2	18.65
`ioctl()`	1.6	0.89	4.27	117.15	116.26	130.37
`_newselect()`	14.8	43.27	17.14	40.85	-2.41	-0.06
`read()`	34.16	1.03	36.9	3.29	2.26	2.19
`close()`	0.01	0.1	0.0	N/A	N/A	N/A
`poll()`	0.0	N/A	0.0	N/A	N/A	N/A
`open()`	0.02	0.9	0.02	684.62	683.72	757.37

Table 5: PROSPECT Slowdown

therefore requires root privileges. In contrast, the PROSPECT server can be run as any user on the target system. It is however recommended to run it as root, simply to ensure that all devices are accessible. Through PROSPECT, the investigated process inherits the device access permissions from the server. As a result, it could be possible for an investigated process to access devices even though that would not be possible under normal circumstances. This property is not necessarily a limitation per se, as it constitutes an additional possibility to influence system behavior during analysis.

With regard to the high slowdown for unrestricted `ioctl()` calls, our implementation still provides room for improvement. For instance in future implementations, instead of querying the */proc* file system, we could implement a more efficient mechanism to minimize execution time.

8. RELATED WORK

When dealing with security analysis on embedded systems, most research approaches use static analysis to achieve their goal. For instance, Khare et al. presented some of the key problems that need to be faced when using static analysis techniques on a large embedded code base [9]. In their work they focus on the static analysis of source code to improve the overall security of embedded systems.

In contrast, Ramakrishnan and Gopal do not require access to source code as their static program analysis techniques run on embedded binaries [22]. However, they do not focus on embedded security or vulnerability discovery.

In [18], Zili Shao et al. introduce a mixed hardware/-software system to check for and protect embedded systems from buffer overflow attacks. Their system works during program execution, but is more focused on vulnerability protection than on vulnerability discovery.

In [21], Sumpf and Brakensiek introduced device driver isolation within virtualized embedded platforms. The approach presented here can be considered the most closely related system compared to PROSPECT. The authors created device drivers with a generalized interface to provide homogeneous access for virtual machines. In contrast to PROSPECT, however, the implementation of the driver must be known. Furthermore their system is limited to L4 microkernels and not suitable for unknown peripheral devices.

9. CONCLUSION

PROSPECT turned out to be a valuable tool that enabled us to conduct a full-scale dynamic security analysis of a widely used fire alarm system. Without PROSPECT, dynamic analysis would have been infeasible due to the limitations of the fire alarm embedded system. We believe that PROSPECT's approach has a high practical impact. It allows to overcome the limitations of static analysis that are common for embedded vulnerability discovery. The general concept is applicable to a wide range of embedded systems, including smart phones or field level SCADA components.

10. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the Austrian Research Promotion Agency (FFG) under grants 836276 (SG2), 834005 (Fire-IP) and the European Union Seventh Framework Programme under grant agreement n. 257007 (SysSec). We would like to thank the anonymous reviewers for their helpful feedback and improvement suggestions. We would like to thank Trustworks [12] for providing valuable insights and tools that made this research possible.

11. REFERENCES

[1] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 97–106, 2011.

[2] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 427–430, 2011.

[3] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 47–56, Washington, DC, USA, 2001. IEEE Computer Society.

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394, 2012.

[5] W. Drewry and T. Ormandy. Flayer: Exposing application internals, 2007.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[7] S. Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 4490–4494, 2011.

[8] M. Kermani, M. Zhang, A. Raghunathan, and N. Jha. Emerging frontiers in embedded security. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, pages 203–208, 2013.

[9] S. Khare, S. Saraswat, and S. Kumar. Static program analysis of large embedded code base: an experience. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 99–102, New York, NY, USA, 2011. ACM.

[10] P. Koopman. Embedded system security. *Computer*, 37(7):95–97, July 2004.

[11] B. Liu, L. Shi, Z. Cai, and M. Li. Software vulnerability discovery techniques: A survey. In *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, pages 152–156, 2012.

[12] Trustworks KG. *http://www.trustworks.at (retrieved 2013-04-17)*, 2013.

[13] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[14] S. Parameswaran and T. Wolf. Embedded systems security-an overview. *Design Automation for Embedded Systems*, 12(3):173–183, 2008.

[15] F. Project. Filesystem in userspace. *http://fuse.sourceforge.net/ (retrieved 2013-04-17)*, 2013.

[16] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331, 2010.

[17] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[18] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao, and E. H.-M. Sha. Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. *IEEE Transactions on Computers*, 55(4):443–453, 2006.

[19] K. V. Shibu. *Introduction To Embedded Systems*. McGraw-Hill Education, 2009.

[20] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 635–647, New York, NY, USA, 2009. ACM.

[21] S. Sumpf and J. Brakensiek. Device driver isolation within virtualized embedded platforms. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5, 2009.

[22] R. Venkitaraman and G. Gupta. Static program analysis of embedded executable assembly code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 157–166, New York, NY, USA, 2004. ACM.

[23] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *Security Privacy, IEEE*, 10(5):68–70, 2012.

[24] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security Privacy, IEEE*, 5(2):32–39, 2007.