Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation

Markus Kammerstetter and Daniel Burian

Secure Systems Lab Vienna, Automation Systems Group
Institute of Computer Aided Automation
Vienna University of Technology
Vienna, Austria
Email: {mk,dburian} @ seclab.tuwien.ac.at

Wolfgang Kastner

Automation Systems Group
Institute of Computer Aided Automation
Vienna University of Technology
Vienna, Austria
Email: k @ auto.tuwien.ac.at

***Abstract*—Today, interconnected embedded devices are widely used in the Internet of Things, in sensor networks or in security critical areas such as the automotive industry or smart grids. Security on these devices is often considered to be bad which is in part due to the challenging security testing approaches that are necessary to conduct security audits. Security researchers often turn to firmware extraction with the intention to execute the device firmware inside a virtual analysis environment. The drawback of this approach is that required peripheral devices are typically no longer accessible from within the Virtual Machine and the firmware does no longer work as intended. To improve the situation, several ways to make the actual peripheral devices accessible to software running inside an emulator have been demonstrated. Yet, a persistent problem of peripheral device forwarding approaches is the typically significant slowdown inside the analysis environment, rendering resource intense software security analysis techniques infeasible. In addition, security tests are hard to parallelize as each instance would also require its own embedded system hardware. In this work, we demonstrate an approach that could address both of these issues by utilizing a cache for peripheral device communication in combination with runtime program state approximation. We evaluated our approach utilizing well known programs from the GNU core utilities package. Our feasibility study indicates that caching of peripheral device communication in combination with runtime program state approximation might be an approach for some of the major drawbacks in embedded firmware security analysis but, similar to symbolic execution, it suffers from state explosion.**

Keywords–Embedded Systems; Security Analysis; State Explosion; Program Slicing; Virtual Machine Introspection.

I. INTRODUCTION

The widespread use of embedded systems in security critical environments calls for better security testing techniques. However, testing embedded system firmware in its native environment imposes severe restrictions. Embedded systems can often be interfaced over debugging interfaces such as JTAG (Joint Test Action Group) or serial communication, but they typically only provide very basic debugging functionalities insufficient for more powerful security analysis techniques based on dynamic instrumentation. A possible solution to these problems is to create a VM (Virtual Machine) that emulates the entire embedded system. Since only the most common hardware is emulated by existing emulators, such as QEMU, real world embedded devices may require implementing additional peripheral device emulators. Yet, extending a VM with peripheral devices can not only be too time consuming for a resource constrained embedded security audit, but the information on the internals of these peripherals might not be available in the first place. Ultimately, this renders the emulation based approach infeasible in many cases. Previous work [3, 10] showed how peripheral devices can be transparently connected to a VM. This allows the embedded system firmware to run inside an emulator as if it were running on the original hardware with the peripheral devices directly attached. The extracted system firmware can thus be inspected outside its original system environment. The drawback of the peripheral device forwarding approach is the typically significant slowdown of device communication and the lack of possibilities to parallelize slow analysis runs or to leverage snapshots in presence of external peripheral device states. Since typical security testing techniques such as fuzz testing are highly repetitive in nature, in this work, we evaluate an approach utilizing caching of peripheral device communication in combination with runtime program state approximation. Our approach could ultimately render existing dynamic firmware security analysis techniques more powerful by enabling functions such as snapshotting, test parallelization or testing without physical access to the embedded system. We show that the challenge is not the caching itself but the sufficiently accurate approximation of the embedded program state to decide which peripheral device response in the cache needs to be returned to the firmware under test. We address this problem with runtime program state approximation and show that, similar to symbolic execution, the approach suffers from state explosion. Specifically, the contributions presented in this paper are as follows:

- We present a peripheral device caching approach for embedded security testing.
- We present a state variable detection heuristic allowing runtime program state approximation as key to peripheral device communication caching.
- We evaluate the feasibility of our approach with programs from the GNU core utilities and show that it might be usable to address persistent drawbacks in embedded firmware security analysis in the future.

 ISBN: 978-1-61208-493-0

The remainder of this paper is organized as follows. Section II provides an overview of related work. In Section III, we explain how peripheral devices are typically accessed from within an embedded operating system and describe why these devices are a challenge for current embedded system security testing methods. In Section IV, we present our peripheral caching approach leveraging runtime program state approximation which is described in Section V. The results of our feasibility study are presented in Section VI. The conclusions and suggestions on further work can be found in Section VII.

II. Related Work

In previous work, at least two different peripheral device forwarding approaches have been implemented. In [10], Zaddach et al. presented the Avatar framework allowing existing tools such as the QEMU emulator or symbolic execution tools to be connected to embedded target systems. Based on memory mappings, their system can forward peripheral device access from the emulator to the corresponding memory region of the peripheral device on the target embedded system. Similarly, Kammerstetter et al. presented the PROSPECT framework [3], an operating system centric approach that forwards peripheral device accesses from within the kernel in the VM to a stub on the embedded target device via a network connection. In addition to peripheral device communication forwarding, Koscher et al. presented SURROGATES [6], a system that uses Field Programmable Gate Arrays (FPGAs) to speed up the connection between the forwarding system (i.e., Avatar or PROSPECT) and the embedded hardware itself. In contrast, our work does not focus on the peripheral device forwarding techniques themselves, but instead adds a peripheral device communication caching layer in between the VM and the target device. We thus aim to simplify embedded security testing by enabling powerful mechanisms such as snapshotting, parallelization or testing without the analysis environment being connected to the real embedded system. The concept underlying our caching heuristic is related to the problem of program slicing where our peripheral caching system identifies states of the program slice that deal with peripheral hardware access. In general, program slicing typically focuses on source code and has been broadly covered by Weiser et al. [9], Korel et al. [5], Frank Tip [8] and Binkley et al. [1]. More recently, Kiss et al. [4] and Cifuentes et al. [2] also covered the problem of slicing binary executables. Considering the work on binary slicing, our cache heuristic is loosely related as the cache needs to identify states in a program slice based on the runtime environment of the process. We thus aim at identifying individual states in a program slice of the running process without extracting the whole program slice.

III. Peripheral Device Access

By leveraging peripheral device forwarding, medium to large scale embedded systems can be analyzed for security vulnerabilities. Within this work, we exemplary focus on embedded systems that utilize Linux on a MIPS architecture such as routers or Cyber Physical System (CPS) components. These systems are typically composed of a System-on-Chip (SoC) containing a processor, ROM, SRAM and I/O Controllers. The I/O Controllers are used to connect the SoC to external components such as DRAM, flash memory or peripheral devices (see Figure 1). Depending on the embedded system use cases, connected external peripheral devices are often customly designed by system manufacturers and can range from simple sensors and actuators to complex modules such as communication interfaces or security modules.

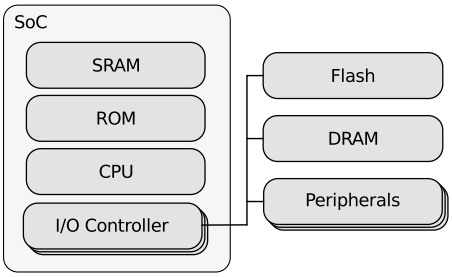

Figure 1. Typical Embedded System Hardware.

A. Challenges in Embedded System Security Testing

The security of embedded systems can be tested in several ways. The manufacturer of an embedded system typically has very detailed information about all components within the system and can thus resort to techniques such as whitebox security auditing and source code security analysis. Embedded systems often provide JTAG or serial console access allowing developers to access the running system. Depending on the specific implementation, these interfaces can provide a varying range of device access ranging from simple status readout to full dynamic system analysis. If the embedded system does not already provide tools for dynamic system analysis, the tester may be able to install necessary tools via an exposed debugging interface. However, embedded systems are typically resource constrained and tailored to a specific task. Without the resources to run additional software like debuggers on the system, dynamic analysis on the device itself is often infeasible. In addition, the operating system kernel may be tailored to the specific use case of the system with debugging or system analysis features stripped to reduce hardware requirements and thus production costs. Whenever dynamic security analysis on the embedded system is not feasible, analysts typically aim at extracting the firmware from the device for further investigation. This can either involve static analysis techniques on the firmware with its well known limitations [7], as well as dynamic analysis approaches utilizing debugging interfaces such as JTAG or VM emulation. At this point, the challenge arises that embedded systems typically make extensive use of peripheral devices that are typically not available from within the VM. The analyst thus needs to resort to peripheral device forwarding frameworks such as Avatar or PROSPECT that have limitations on their own. Specifically, forwarding peripheral device communication is typically impeded by a significant slowdown and a lack of possibilities to parallelize slow analysis runs as each testing instance would require its own connected embedded target system.

B. Communication with Peripheral Devices

On UNIX systems such as Linux, peripheral devices are accessible via system calls that are handled by the kernel which in turn uses device specific hardware drivers for the actual device communication (Figure 2). In our test environment, the peripheral devices are represented as character devices. Depending on which commands the device driver supports, a user space program with the right permissions can thus access these

 ISBN: 978-1-61208-493-0

devices with system calls such as `open`, `read`, `write` or `close`. To enable dynamic analysis on an otherwise resource constrained system, we utilize the PROSPECT framework [3]. The Linux kernel and all encompassing software running on the system are extracted from the embedded system and moved into an emulator such as QEMU.

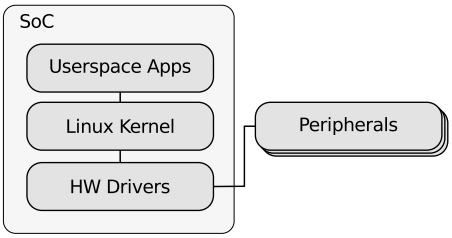

Figure 2. Typical Embedded System Software Stack.

To allow programs in the VM to communicate with the peripheral devices in a way similar to the original embedded system, PROSPECT replaces the embedded system software on the original hardware with a server stub that forwards all device communication over a network connection to the peripheral devices. We thus tunnel all device communication from the VM to the peripheral devices via a network connection such as TCP/IP over Ethernet. This allows us to run the embedded system software inside the analysis environment and thus enables the use of resource intense analysis techniques. Although the analysis environment typically provides significantly more system resources such as file system space, CPU speed or RAM, previous research showed that due to the peripheral device forwarding [3] most device communication will be significantly slowed down. Besides, another drawback is that each VM will require a dedicated set of embedded system hardware.

IV. Caching Peripheral Device Communication

Considering security testing techniques such as fuzz testing, tests are typically highly repetitive and focused on very specific (i.e., security critical) code regions in the firmware. Triggered by each of those very similar test cases, the firmware of the embedded system performs the very same communication actions with its peripheral devices over and over again. For instance, consider a Real Time Clock (RTC) peripheral device that would be queried by the embedded firmware each time a network packet is received. Although the security analyst might only target the network packet handling code in the firmware with the fuzz tester, the peripheral device communication to the RTC would still need to be carried out as otherwise the firmware would stop to function and could not be tested. As long as the values returned from the RTC allow the firmware to continue its normal execution, it is not necessary that the returned values are actually correct. Although two subsequently read timestamps should represent an amount of time that has passed between the successive reads, the functionality of the firmware during the focused fuzz tests will in most cases not be impeded by the fact that the time itself is not correct. By adding a peripheral device communication cache between the analysis environment and the embedded system, the repetitive device communication actions could be stored so that during the highly similar test cases valid device responses can be served from the cache. Ultimately, this would enable very powerful supporting technologies such as snapshotting, parallel testing or even testing without the embedded system attached.

A. Caching Strategies

In the first step, we implemented a cache between the PROSPECT driver and its stub on the target device (Figure 3). For each peripheral device interaction, the cache receives the following information:

- Process Id (PID) and Thread Group Id (TGID)
- Name of the peripheral device
- Command type and command data

When the cache receives a command, it has to decide between two options:

1) Cache hit - An appropriate device response is already in the cache. The cached response is returned to the program without querying the actual device.

2) Cache miss - The cache has not stored a suitable device response. In this case, the cache first needs to bring the hardware into the state it would normally be before this request. This is done by resetting the hardware and replaying all communication that the requesting program performed until this point. The approach can thus forward the new command to the peripheral device, store the new reply and forward it to the VM. This means that the cache needs to retain information not just about the commands and their replies, but also about the previous command history for each VM.

The main challenge is to find a strategy that can be applied to decide whether for a specific firmware program state a valid peripheral device response is already in the cache. We explored several strategies and describe them in the following:

1) Choosing Responses by Command: Very simple devices may be cached by command. To do so, the device must either be stateless, or the device's state must be deducible from the command. For example, if the device is a simple switch that is only controlled by an open and close command, the cache does not need any information other than the command itself to react accordingly. For instance, whenever the cache receives a control request to turn on the switch it could just return the cached response confirming that the switch has been turned on. However, as soon as a single control command can return different responses this approach is no longer applicable. An example where this approach would not work is the above mentioned RTC module which would return a different timestamp value for every read command.

2) Choosing Responses by Command and Command History: An improved strategy is to store information about the previously issued commands to a peripheral device. Based on the command history, the cache can decide if a suitable device response is already in the cache or not. A simplified deterministic example could be a program that reads from a peripheral device representing an incrementing counter with an initial reset. After reset, the program would read continuously increasing counter values (i.e., 1, 2, 3, etc.) on each execution. The read command itself may look the same, but depending on which and how many commands were issued to the peripheral device before, the replies to each command need to be different for every call. If the cache can learn a sufficient amount

 ISBN: 978-1-61208-493-0

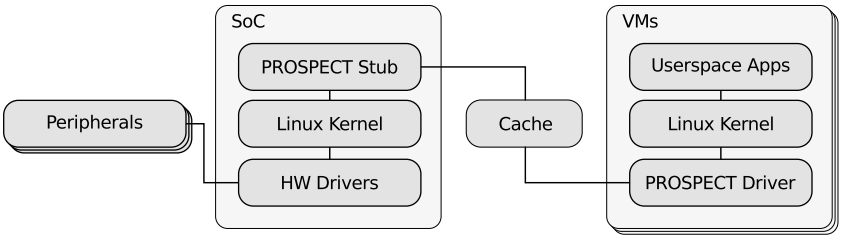

Figure 3. Embedded System Testing Utilizing PROSPECT with an Intermediate Cache.

of requests and replies from the first *training* execution, it can replay the answers every subsequent time the program is executed. The problem with this strategy is that even if the behavior of the peripheral device is deterministic, it becomes insufficient as soon as multiple threads access the same device. In this case, the thread scheduler will cause a different execution order of threads for the same input and the behavior from the perspective of the cache will no longer be deterministic. Since the cache would need to consider all possible thread execution orders to respond to future requests, the strategy quickly becomes ineffective with an increasing amount of program indeterminism. Listing 1 shows an example where two threads cause the mentioned problem by accessing a temperature sensor and a communication interface at the same time.

```
1  def Thread1():
2    while(True):
3      temp = readTemperature()
4      if (temp > max):
5        sendMessage("High temperature")
6      sleep(0.1)
7
8  def Thread2():
9    while(True):
10     statusMsg = getStatusMessage()
11     statusMsg += readTemperature()
12     sendMessage(status)
13     sleep(2)
```

Listing 1. Threading Example with Read-Loop.

3) Choosing Responses by Program State Approximation: A more advanced strategy is to find a heuristic to identify abstract program states reflecting the current position within the program flow. When program execution is started, the program typically makes use of resources such as the CPU or stack memory. We could thus derive a set of relevant CPU registers (i.e., the instruction register, the stack pointer, the general purpose registers) and use this information to determine in which state the program currently resides in. Whenever a peripheral device is accessed (e.g., with a `read` system call), we use the program state to determine whether there is already a known peripheral device response in the cache. If this is not the case, we forward the peripheral device communication from the analysis environment to the real system and cache the response for later use. However, considering typical program constructs such as a loop reading a temperature value (Listing 1), it is very likely that the CPU registers will be identical within the `readTemperature()` function at the call site of the `read` system call for different loop iterations. It is thus necessary to include the program stack into the state computation so that the state of the outer function will be considered as well. However taking the stack memory into account, determining the program states gets much more challenging as it is no longer clear which memory regions are relevant with regard to the peripheral device communication. If the state approximation granularity is too low, many irrelevant memory regions will influence the program state approximation and different program states will be derived for the same peripheral device communication action (*state duplication*). As a result, most of the device accesses would be cache misses. In contrast, if the granularity is too high, we would get wrong cache hits and the program would receive invalid peripheral device responses. In the following, we present the runtime program state approximation approach we took and the results we were able to obtain with it.

V. Runtime Program State Approximation

On an Operating System (OS), the program state can be determined through its allocated memory (i.e., stack and heap), the CPU registers and handles received from the OS kernel (e.g., file handles). However, especially considering binary executables where the source code is not available, determining which variables need to be considered during the determination of the program state is considered to be a hard problem related to program slicing [2, 4]. Since it would not be feasible to deterministically detect exact program states, we implemented a heuristic (Figure 4) that attempts to approximate sufficiently exact program states to use them for our caching approach.

A. System Call Interception and Kernel/VM Hooking

In the first two steps of the heuristic (Figure 4), we need to intercept the systems calls used for peripheral device communication. For each intercepted system call, we need to decide whether the system call is utilized for communication with a device that is forwarded through PROSPECT. Furthermore, for runtime program state approximation we need to have access to the internals of the OS kernel and the program accessing the device as well. This includes the state of the virtual memory at the time of a call, the CPU registers and open file handles. We implemented and practically tested the following methods to obtain the required low-level information.

1) Virtual Machine Introspection (VMI): The first method was implemented by extending QEMU with a Virtual Machine Introspection (VMI) module. VMI has the advantage that any low-level state information from the machine including physical memory or otherwise hard to access kernel internals can not only be accessed and read, but can be modified just the same. An additional advantage is that any introspection logic runs directly on the host machine and not inside the VM, leading to significantly higher performance. Although the VMI approach is very powerful, our VMI module implementation uncovered two major drawbacks. First, due to the low level VMI operates on, important functions inside the kernel such as those providing paging information and memory mapping need

 ISBN: 978-1-61208-493-0

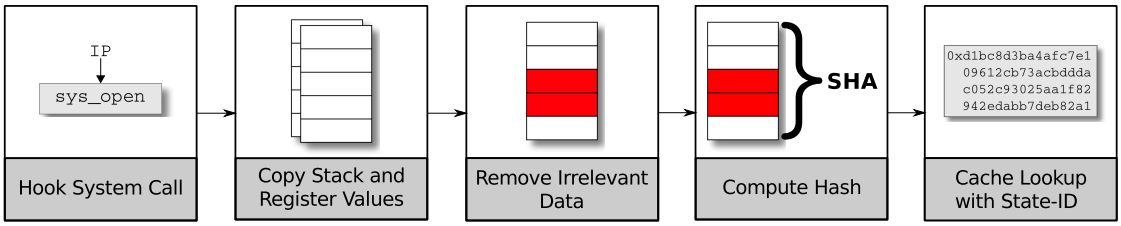

Figure 4. State Approximation Heuristic.

to be reimplemented. Even worse, important offsets to internal kernel structures can be configuration dependent requiring frequent adaptations of the VMI analysis code. Second, to reliably hook system calls, Translation Block Chaining (TBC) needs to be disabled. TBC is an optimization technique the QEMU emulator uses to drastically speed up emulation. Translation blocks are basic blocks of code from the guest system that are translated to the host system architecture. With TBC, these blocks are chained together and cached so that they do not have to be translated again each time the process counter arrives at that specific address. However, due to the caching, the program addresses within these cached blocks are no longer processed by QEMU's TBC lookup logic which ultimately causes our hooks on those addresses to no longer get executed. Disabling the TBC optimization allows reliable VMI hooking but at the same time significantly slows down the emulator.

2) Kernel Module: The second state approximation method was implemented as a loadable Linux kernel module running within the QEMU guest system. Since PROSPECT already performs system call hooking from within the kernel, we extended it with functions to read registers and mapped virtual memory regions of the calling process. Compared to the VMI approach, using a kernel module simplifies access to swapped out pages and kernel structures.

3) File Handles: Each time an `open` system call is used to return a new file handle, the value of the file handle is determined by the operating system kernel. Since the returned file handles frequently differ between executions, we use a file descriptor tracking mechanism. The mechanism places a hook on the `open` and `close` system calls. It can thus track the currently active file descriptors and remove them from the stack region of interest by overwriting the descriptors with zero bytes.

4) Registers: The register content has a central role in our program state variable detection heuristic. While the most important register to be utilized in this case is the instruction pointer, we found that the subset of registers leading to the best results also included the return address, the stack pointer and several general purpose registers.

B. Hash Computation and State-ID Matching

In the last two steps of the heuristic (Figure 4), we compute the SHA-256 digest and use it for cache lookup. The digest is computed on the concatenated stack region of interest and the register set. Using the previously described state variable detection heuristic, we ensure that hash digest results in a granularity that is suitable for cache lookups. The cache lookup is implemented as a large dictionary where the SHA-256 hash value is used as index to a device response data field of arbitrary size.

VI. Results

Our feasibility study shows that our approach works for less complex programs but suffers from the well known state explosion problem for more complex programs. The low complexity programs we tested required less information from stack and registers to correctly determine the program state. However, with growing program complexity, it becomes more challenging to accurately determine a unique state suitable for cache lookups resulting in state duplication and cache misses. Since the number of these duplicates rises exponentially with increasing program complexity, similar to symbolic execution, the approach leads to the state explosion problem. In that regard, the MIPS architecture turned out to be especially challenging due to its standard calling convention and the resulting difficulty of stack frame unwinding. To test our approach, we used programs from the GNU core utilities and treated their file system accesses as peripheral device accesses with our caching approach in between. We tested 3 program classes:

1) Low Complexity Programs:
   For very simple programs such as `cat`, `head`, `sum` and `wc`, our caching approach hardly depends on stack frame information, no heap information is required and only a small subset of the registers is sufficient to correctly determine the program states for peripheral caching. Within a single execution the cache could thus already learn all necessary responses and use them correctly. At that point we were able to completely remove the program's input files and still obtain the identical program flow with our caching approach.
2) Medium Complexity Programs:
   Medium complexity programs such as `expand` rely on dynamic heap memory management. As a result, some of the relevant program states for device access may depend on the information stored at those memory regions. Using peripheral caching for programs like `expand`, the lack of information on heap content led to duplicate states. These could be compensated for by utilizing several training executions until the cache had learned all possible states including duplicates. It also required minor manual adaptations of the considered stack parameters within the heuristic. We believe that this problem can be addressed in future work and the heuristic could be greatly improved by adding proper stack unwinding. Monitoring the heap state would be an advantage, but is not mandatory. Without proper stack unwinding and manual adaptations, medium complexity programs currently present the limit of our approach.
3) Higher Complexity Programs
   Higher complexity programs such as `sort` not only heavily rely on dynamic heap memory management, but

 ISBN: 978-1-61208-493-0

they also store a large amount of relevant state information on the heap. The problem and its possible solution are thus similar to medium complexity programs, but in comparison the number of duplicate states is much higher and can no longer be handled through manual adaptations. We believe that with stack unwinding and dynamic memory allocation monitoring the problem can be improved, but higher complexity programs will remain challenging.

VII. Conclusion and Future Work

Our feasibility study showed that the presented peripheral caching concept could be an approach for some of the major drawbacks in embedded firmware security analysis. When applying typical embedded security testing techniques such as fuzz testing, sufficiently precise caching of peripheral device communication could thus enable powerful features such as snapshotting or test parallelization. After sufficient cache training the firmware can even be tested without requiring physical access to the embedded system. We showed that the problem is related to program slicing and may lead, similar to symbolic execution, to the well known state explosion problem. We created a VMI-based as well as a kernel-module based implementation and tested the feasibility of our approach with programs from the well known GNU core utilities package. Our results show that the peripheral caching approach works for low and medium complexity programs. However, depending on the architecture and the difficulty of stack frame unwinding, the program state approximation can become increasingly difficult. In future work, we're looking forward to port our approach to embedded architectures such as ARM allowing more precise stack unwinding. We believe that this will further increase the precision of the program state approximation so that more complex programs can be addressed with our approach as well. Furthermore, we aim to implement a kernel module/VMI hybrid implementation to benefit from the speed improvements of running the program state approximation heuristic outside the VM while still utilizing the OS kernel insight provided through a kernel module.

Acknowledgements

The research was funded by the Austrian Research Funding Agency's (FFG) KIRAS security research program through the $(SG)^2$ project under national FFG grant number 836276, the AnyPLACE project under EU H2020 grant number 646580, and IT security consulting company Trustworks KG.

References

[1] David W Binkley and Keith Brian Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

[2] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 192–199. IEEE, 1999.

[3] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 329–340, New York, NY, USA, 2014. ACM.

[4] Akos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 118–127. IEEE, 2003.

[5] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[6] Karl Koscher, Tadayoshi Kohno, and David Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, pages 7–7, Berkeley, CA, USA, 2015. USENIX Association.

[7] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, pages 152–156. IEEE, 2012.

[8] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[9] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[10] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 14, February 2014.

 ISBN: 978-1-61208-493-0