2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation

Sentinel: Secure Mode Profiling and Enforcement for Embedded Systems

Paul D. Martin*, David Russell†, Aviel D. Rubin†, Stephen Checkoway‡ and Malek Ben Salem§
*Harbor Labs
Baltimore, MD
Email: paul@harborlabs.com
†Department of Computer Science, Johns Hopkins University
Baltimore, MD
Email: {drusse19, rubin}@cs.jhu.edu
‡Department of Computer Science, University of Illinois Chicago
Chicago, IL
Email: sfc@uic.edu
§Accenture Technology Labs
Arlington, VA
Email: malek.ben.salem@accenture.com

***Abstract*—Embedded devices are designed to cover many possible use cases. In practice only a small subset of features may be used in a given deployment. As devices age, some features turn out to be security risks. We address these problems by creating Sentinel, a secure mode profiler for embedded devices. Sentinel uses a bus tapping interface to derive a partial control flow graph during device execution. This graph represents the subset of device modes actually observed during use. The control flow graph is generated without any prior knowledge of the device or its software and constitutes a security profile which can be used to audit device execution in order to detect attacks. The profile can be easily enforced by existing bus monitors with minor modifications.**

***Index Terms*—Computer Security; Internet of Things**

I. Introduction

With the advent of the Internet of Things, today's embedded devices have become increasingly connected [1–3]. One of the major benefits of this connectivity has been realized in industrial networks, where device measurements can now be taken remotely. Gone are the days when technicians would carefully transcribe analog readings produced by disparate equipment onto paper charts. Modern industrial networks leverage technology in order to minimize the number of times a technician must perform a manual reading or use paper records. Many industrial networks contain a mix of general-purpose computers and mission-critical embedded devices [4–8].

These devices are often installed directly by an agent of the manufacturer or by field technicians specifically trained by these agents. There are numerous examples of embedded devices containing special modes that are meant to be used only by these agents during setup [9, 10]. Unfortunately, manufacturers still produce devices that do not have secure default configurations [11–13] and in many cases these modes can also be used by attackers to maliciously reconfigure and attack the device [12, 13].

In addition to being vulnerable to remote attacks like any other network-facing device, many embedded systems are vulnerable to physical attacks. Industrial equipment must often be deployed in the field. In such an environment, the equipment owner typically has little control over who interacts with the devices. However, until recently, most manufacturers have focused on reliability and usability rather than on security. There are many potential negative consequences of an attacker tampering with industrial equipment: the attacker could degrade performance, tamper with measurements, disable functionality or, in some cases, even cause direct bodily harm [14, 15]. Physical security in industrial networks is a serious problem that does not always receive the attention that it deserves. When we consider physical attacks in conjunction with remote attacks, we see that many embedded devices have enormous attack surfaces.

To further complicate matters, many embedded devices are designed to have extraordinarily long lifespans. For example, a medical infusion pump model might be supported by its manufacturer for 10–15 years [16]. This creates several problems. As security techniques constantly improve, legacy devices are often left behind. In some industries, such as healthcare, embedded devices may not even be able to receive software upgrades without the update first going through a lengthy and expensive review process [17]. Due to this review process, manufacturers sometimes issue recommendations against continuing to use a particular feature in lieu of a firmware update that actually removes the mode [18]. Furthermore, industrial devices are designed with longevity in mind and they are often designed to be sold to a broad spectrum of customers. Devices often contain a wide variety of modes and features in order to meet the requirements of all possible customers. Most customers will only use a subset of these possible features and modes.

In many cases actual usage of the device may fit a very narrow profile. Despite the extensive work looking at securing

0-7695-6387-2/18/$31.00 ©2018 IEEE
DOI 10.1109/IoTDI.2018.00020

IEEE computer society

embedded devices against exploits [19–29] there has been little work on how to build and enforce security profiles that cover a device's typical usage patterns, and how to disable insecure device features without requiring a firmware update.

This new work would be a useful complement to existing work on defense against exploits, as it would allow a manufacturer to dramatically reduce the attack surface on an embedded device by allowing the device to be locked down to a limited profile during deployment. For example, on devices with a configuration mode, it would be useful to disable [1] the configuration mode while the device is deployed in the field if the device should only be reconfigured when it is offline [30]. Similarly, on devices that use interfaces for debugging purposes, it would be useful to disable these interfaces when the device is not being debugged [31–33]. On legacy devices with telnet access, it may be useful to disable telnet altogether [34–36]. Note that these three examples all come from vulnerabilities reported in actual industrial embedded devices. Existing exploit mitigation techniques would not adequately address these issues but a device for building and enforcing device profiles as described in this work would.

In many situations firmware updates are not available for a given device or they cannot be feasibly pushed to devices in the field. For example, some infusion pumps that we examined contain socketed ROMs. Other deeply embedded devices may not be firmware-updatable at all. If there is no remote firmware update functionality in a given device then the firmware needs to be upgraded by a field tech. This becomes a problem if the firmware needs to be updated by hand every time a new vulnerability is discovered, especially since these devices may have 10-20 year lifespans. In these cases, having the ability to disable vulnerable modes (such as telnet modes found in devices built in the early 2000s) without requiring a recall would greatly improve the security posture of the underlying device, even in the face of an uncertain future security landscape.

To address these problems we create Sentinel, a secure device profiler for non-SoC (System-on-Chip) based embedded systems with external memory busses. Sentinel significantly increases the physical security of target devices by using a bus tapping interface to derive a partial control flow graph representing a subset of functionality of the attached embedded system. This control flow graph is built from device execution traces taken while running the device through the desired functions. Anything in the control flow graph is considered an allowable action within the device's security profile while any action not in the security profile is considered a security violation. Sentinel builds its partial control flow graph by monitoring the memory bus directly. It is therefore a passive observer to normal device operations. Its profiles are enforced using one of the existing snooping-based runtime enforcers. Because of its passive design, Sentinel is able to build profiles without interfering with the normal operation or timing of the attached device. Similarly, its profiles can be enforced without any additional runtime overhead. Furthermore, Sentinel is robust even in the face of lossy data.

Sentinel uses a bus tap to extract samples of partial control flow graphs directly from the address bus of a target device at runtime. These samples are combined by the device profiler to recreate the partial control flow graphs corresponding to a set of desirable device behaviors referred to as security profiles. The security profiles are designed to be enforced by existing bus snooping-based control flow integrity techniques [19–21] with only minimal modifications. Sentinel can thus be thought of as a combination of a novel bus tap and offline device profiler combined with one of the snooping-based runtime enforcers in existence today.

We outline a high-level design for an execution-based embedded device profiler. We prove its feasibility by using it to build a security profile of a particularly important type of embedded device–a medical infusion pump. We test the utility of the profile by using it to audit an execution trace taken during a physical attack on the device. We further outline how we solve difficult problems such as accounting for interrupt and exception handlers and anticipating instruction prefetches before looking at how our profiler handles lossiness using sample size and false positive rate. Finally, we consider additional uses for Sentinel beyond our novel security profiling technique.

Threat Model. This work is concerned with attackers that already have some access to an embedded system. The attackers may be physical or remote. Their goal is to attempt to access a feature or mode of the device that is not within its use profile without detection.

We are interested in embedded devices that perform important functions and are complicated enough to have multiple device modes. These include things like infusion pumps and other medical devices, certain types of industrial control systems, automotive control systems and airplane control systems. Additionally, many of these types of embedded devices use ROMs [37–42]. Such devices often contain modes that may be necessary for configuration but that should not be used in deployment [30–36]. An attacker may wish to exploit such a weakness in the device in order to maliciously reconfigure or control the device or tamper with its operation.

Note that we are not concerned with a user physically disabling a Sentinel. In order to do so the user would need to disassemble the embedded device and desolder the Sentinel from the PCB. If an attacker is going to this extent to tamper with the device then the attacker could also perform numerous other malicious modifications to the device. Instead, Sentinel is supposed to sit inside of a device's casing and detect when a user uses the normal device interface to enter an undesirable device mode. This interface may be local or remote.

This work is also not concerned with varied parameter attacks, in which an attacker attempts to corrupt a value stored in memory in order to cause a device to perform an undesirable, but legal, action in a profiled mode. Varied-parameter attacks are certainly a problem, but they are a different problem than the device mode problem that we explore in this work and

[1]Disabling a device feature is one of several possible mitigation techniques that a Sentinel could be made to support. These techniques are discussed in more detail in Section IV-E.

thus they require a different solution.

Our Contribution. This work proposes the following significant and novel contributions:

1) Dynamically builds partial control flow graph corresponding to device features and modes without any knowledge of the underlying source code or firmware
2) Audits execution traces to ensure that the device operated within a given security profile
3) Considers the relationship between number of samples and false positives when accounting for lossiness

II. Related Work

There has been much previous work looking at memory bus monitoring in embedded devices [19–21, 43–46]. Some of this work has used bus monitoring in conjunction with other techniques to add security to insecure embedded devices [20, 43, 44] including by using a modified compiler to output a full control flow graph to be used in conjunction with a bus monitor to enforce control flow integrity [19–21].

All control-flow related bus monitoring work that we know of has looked at using the full control flow graph to detect and prevent remote exploits rather than to limit device functionality. In contrast, our work dynamically builds partial control flow graphs that encapsulate specific device functionality. These partial control flow graphs can be enforced as security profiles at runtime or they can be used to audit device execution in order to ensure that an attached device is operating according to its expected run-time profile.

A. Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors

This work [19] is a simulated design for a bus monitoring circuit that can enforce control-flow integrity on embedded devices in real-time. The design enforces inter-procedural control flow, intra-procedural control flow, and instruction stream integrity. Inter-procedural control flow is accomplished by storing all procedures in a function call graph that is translated into a finite-state machine. Whenever the monitor observes a function call it checks to see whether the call corresponds to a valid state transition. Intra-procedural control flow is accomplished through the use of a basic block table. For each basic block, its two possible successor addresses are stored (the address of the next basic block depends on whether or not the branch was taken). Instruction stream integrity is accomplished by also storing the hash of each basic block in the basic block table, hashing the instruction stream for the basic block at run-time, and ensuring that the observed hash matches the hash stored in the table. The described data structures are stored in an *enhanced executable* and are loaded into a hardware monitor at run-time.

B. A Watchdog Processor to Detect Data and Control Flow Errors

This work [21] proposes a design for a hardware watchdog processor for Motorola M68040-based embedded devices. The watchdog is used for integrity and reliability purposes rather than for security. The watchdog is implemented as a custom circuit that is capable of detecting faults caused by radiations and electromagnetic interferences. These sorts of faults can cause two types of errors: data errors and control flow errors. The proposed watchdog processor can thus detect both types of errors. The watchdog protects against data errors by implementing a bus protection strategy based on Automatic Repeat Request. The watchdog protects against control flow errors by calculating a signature for each *branch free block* in the entire binary and storing the signatures for all blocks offline. At runtime the signature is recomputed and compared to the stored signature.

C. Vigilare: Toward Snoop-based Kernel Integrity Monitor

Vigilare [44] is a hardware kernel integrity monitor designed to facilitate integrity checking on operating system kernels (and hypervisors in particular). Existing hardware-based integrity monitors used sampling-based bus traffic monitoring schemes due to implementation-level difficulties that this work purported to solve. In contrast to the snapshot-based approaches, Vigilare implements a real-time approach capable of monitoring all bus-traffic rather than only a limited amount. Vigilare loads the addresses of important kernel symbols within static regions from the System.map file and verifies the integrity of the data at these addresses during device runtime by capturing write operations to these addresses.

III. Background

Background knowledge of several key concepts will aid in understanding our solution to the problem of building partial control flow graphs and auditing execution traces.

A. Address Space Layout in Embedded Systems

The memory bus is used to transfer data between the CPU and a memory device. It is composed of three smaller buses: an address bus, a data bus, and a control bus. The address bus specifies the logical address of the memory to access. The data bus is used to communicate data to or from the memory device. The control bus is used to tell the memory controller what type of bus operation is to occur, as well as when an address or data is latched on the address or data bus, respectively.

Many embedded systems store their firmware on a ROM chip that is directly mapped into the address space rather than on a secondary storage such as a hard drive or solid state drive [47]. In such a system the CPU can execute code out of the ROM by directly loading instructions from its memory addresses. Additionally, these systems typically contain a RAM that is also mapped to the address space. Typically RAM and ROM devices use an internal addressing scheme of linear addresses starting at address 0x0. Thus, linear addresses seen by the CPU must be translated into physical addresses used to access the data in the physical memory device. A memory controller translates linear addresses into physical addresses.

B. Instruction Prefeteching

Almost all modern CPU architectures implement some form of instruction prefetching [48] in order to keep the CPU pipeline filled with instructions. When the CPU pipeline is empty the CPU must wait for its next instruction to be fetched from main memory or from cache. This may be a relatively time consuming process. Instruction prefetching helps to minimize how often the CPU must wait on instructions to be fetched. One of the earliest architectures to implement instruction prefetching was the Intel 8086, which could linearly prefetch up to six bytes of instructions [49]. More modern architectures implement significantly more advanced prefetching algorithms to provide branch prediction [50] and branch target prediction [51].

C. Interrupt Handling

Most CPU architectures support facilities for handling exceptions as well as environment-triggered inputs, which may be triggered in an unpredictable fashion [52]. Such constructs are signaled to the CPU externally through an interrupt controller or internally through a software interrupt. The CPU, upon detecting an exception or an interrupt, will execute a corresponding handler for the event. The list of all possible handlers is typically stored as a jump table somewhere in system memory. The precise location of this interrupt table is architecture–dependent. When interrupts are triggered the system will choose what to do based on a combination of architectural specification and system state. The system may ignore certain interrupts or delay their handling, or it may immediately halt execution and jump to the interrupt handler. In many cases the interrupt handler will return to the previously-executing code upon completion. In some cases the interrupt handler may not return at all.

D. Control Flow Integrity

Control flow integrity is a defense technique that can protect a system against control flow hijacking attacks such a buffer overflows [53] and return-oriented programming [54]. Control flow integrity typically protects against such attacks with the use of a control flow graph generated by the compiler at compile time. The control flow graph is a graph of all basic blocks and transitions in a program. This graph is used to generate a state machine. The program execution is then monitored and all control flow transitions are checked in accordance with the generated state machine. If any observed control flow transition does not correspond to a transition in the control flow graph, then the monitor will trigger a security violation and halt the attack.

IV. Design

Sentinel is realized as a compound design consisting of two orthogonal components: a bus tap and a device profiler. A fully-deployed Sentinel system would use one of the existing runtime enforcers defined in the related work section to enforce a profile. The Sentinel bus tap is connected to an embedded system's address and control buses. The bus tap forwards the start and end addresses of all basic blocks fetched by the CPU to the device profiler. It accomplishes this by monitoring all instruction fetches. If it detects a jump between basic blocks it inherently forwards the jump source and target to the device profiler. The jump target represents the first address in a new basic block and the jump source represents the last address in the previous basic block.

The device profiler receives the list of basic block addresses, starting with the address of the first basic block fetched by the CPU, when the device initially boots. It uses these basic block addresses to construct a full control flow graph of the observed device execution. Thus, the device profiler receives device execution traces and constructs a control flow graph from them. This control flow graph is, by definition, sparse because any basic block start address is associated with a single basic block end address, and any basic block has exactly two possible successors based on whether the jump at the end of the block is taken. Thus, the device profiler stores this control flow graph as an adjacency list. Because execution traces are lossy, the profiler can be run on the concatenation of multiple execution traces in order to construct a more complete profile. This is described in more detail in the following sections.

This partial control flow graph represents a *security profile*. Any device modes or features that were accessed during the execution trace will be a part of the security profile, and therefore will be considered allowable accesses during subsequent audits or enforcement. Conversely, any features or modes not in the security profile will be considered anomalous accesses. The Sentinel prototype that we have constructed can audit arbitrary execution traces offline in order to determine whether or not the execution trace violates a given security profile. Combined with the control-flow based real-time bus monitors proposed in the prior work [19–21] one could enforce Sentinel's security profiles on embedded devices in real-time.

A fully-deployed Sentinel (complete with enforcer) could either be integrated into existing designs internally or it can be attached to existing hardware externally. Because Sentinel is an open design it can be easily modified to fit a variety of use cases. In our embodiment, Sentinel is designed to be easily integrated into embedded systems that meet certain architectural requirements. While it may be possible to modify the Sentinel design to fit into other types of embedded systems, we will restrict our discussion to cover only the types of systems into which Sentinel can currently be integrated with no modifications.

A. Architectural Requirements

Sentinel can be integrated into any system that meets its architectural requirements. The Sentinel device profiler is architecture agnostic as it relies only on the abstraction provided by the bus tap in order to build the control flow graph. The Sentinel bus tap, on the other hand, can only be integrated into embedded devices that meet several basic architectural requirements. In practice these requirements are minor and most devices that are not system on chip (SoC) should meet them. Even SoC devices with additional external memory could

possibly be made to work with the Sentinel bus tap in a more limited capacity.

1) External Memory Bus: The bus tap must translate the raw electrical signals that are sent across the memory bus into an infinite stream of addresses. Therefore, in order to be a candidate for Sentinel integration, an embedded system must fetch its instructions from external memory banks such as ROM and/or RAM. If an embedded system were to fetch its instructions from internal memory banks instead (as would be the case with a SoC), Sentinel would have no way to monitor the address and control buses. Note that this limitation does not extend to caches. Sentinel supports caches as it monitors the data path from the CPU to the system memory. Thus, data will be observed by the Sentinel bus tap as it fills the cache. Sentinel need not be aware that data is subsequently accessed from the cache.

In some types of embedded devices, the internal memory on the SoC is limited and is only used to store a bootloader or some limited subset of the code. The rest of the code will still be stored in external memory. In these cases the Sentinel bus tap is likely to work without issue. Since the device we have used for our demo is not SoC-based, testing these architectures is left to future work.

2) CPU Address Bus Control Pins: The bus tap must also be able to differentiate between bus operations in order to determine when a bus operation corresponds to an instruction fetch. In many architectures [55–58] the CPU signals the type of bus operation requested to the memory bus controller using a control bus. The bus tap uses this control bus to differentiate an instruction fetch from any other type of memory bus operation. While Sentinel's bus tap implementation is configured to interface with the x86 architecture for the purposes of our demo, it could be trivially modified to interface with any other architecture that exposes memory bus control operations through external signals.

B. Integrating Sentinel into Embedded Devices

There are numerous ways in which one could realize the Sentinel architecture in an actual product design. In one embodiment, we envision the Sentinel bus tap and device profiler being packaged together on a custom ASIC along with an enforcer. This ASIC would sit between the CPU and memory bus controller. The manufacturer would include a *profiler mode* switch inside the device. When set to *profile mode* the ASIC would automatically build a new security profile with the desired functionality and it would store the security profile on an external flash. When set to *enforce mode* the ASIC would enforce the stored profile on instruction fetches in real-time. This would protect the device against physical attacks (such as accessing "locked" modes) as well as against exploits (such as return-oriented programming attacks).

C. Methods of Integration

Sentinel can either be externally wired to existing devices or it can be integrated into device designs as a custom ASIC. To connect Sentinel to an existing device it must be wired to the address bus. Alternately Sentinel is designed to be easy to integrate into new revisions of the printed circuit board (PCB) layout of devices already in production or to be incorporated into the initial revision of new device designs. This flexibility allows a design to be retrofitted with Sentinel at minimal cost. Furthermore if a device design already uses a custom ASIC as a memory bus controller, as was the case in several devices that we examined, then Sentinel can be integrated directly into this ASIC. Otherwise, Sentinel can be placed between a CPU and a memory bus controller in order to monitor and audit bus operations.

D. False Positives

False positives are a possibility with any security system such as Sentinel. It is always possible that a device may not demonstrate some functionality that is a legitimate part of device operation but that occurs infrequently enough to be missed when building a profile. For example, some embedded devices check for updates or perform measurements at extremely infrequent intervals (such as once per month). In these case, Sentinel would report a security violation when there is no legitimate cause for alarm.

While this may be a concern, the possibility for false positives hardly makes a device profiler useless. Although intrusion detection systems generate numerous false positives, many network administrators still view IDS technology as an important and necessary component in a network security stack. In fact, in many cases, the most important modes that are to be protected are the easiest to profile. For example, infusion pumps administering therapy are often designed to disallow other operations while therapy is in progress [59].

Thus, there is little chance that an unexpected event will occur in such a mode, meaning that the mode is able to profiled without the risk of false positives. As an aside, if one were to use Sentinel with a device with such strict safety requirements, the preferred alert behavior would be to trigger a loud and audible alarm rather than to halt therapy immediately. This is actually the default behavior for the Alaris infusion pumps that we examined.

Furthermore, in the case that a profile has been correctly built to cover all desired device functionality, Sentinel is intended to never report a security violation when none exists. That is, Sentinel is designed to be robust against false positives in all cases except for errors in building a device profile. In the case that an existing profile does not adequately cover the full control flow graph of device functionality, it is easy to augment an existing profile with a partial control flow graph covering newly observed behavior.

If false positives are a particular concern for a given application, one could easily modify Sentinel's device profiler in order to exclusively profile device modes rather than to inclusively profile them. These profiles, which would only contain undesirable device modes, could be built by profiling the undesirable mode in addition to the device's normal operation. Basic block addresses observed to be in the

undesirable mode but not in the normal operation modes would would indicate that the device is in an invalid mode.

Finally, it is important to note that an important property of the devices that Sentinel has been designed to secure is longevity. Devices such as medical infusion pumps often have 10-20 year lifespans. After a long enough learning period it is highly likely that a device profile will be fully inclusive and thus that Sentinel will not report any false positives. The longer the device is profiled the less likely a Sentinel is to report a false positive. Thus, as a Sentinel-enabled device ages (and thus as the likelihood of discovered vulnerabilities significantly increases) the likelihood of a false positive actually decreases.

E. Failure Modes

Sentinel is designed to be flexible enough to be configured to meet a variety of applications. The specific safety and reliability needs of the particular applications could tailor its response to security alerts. If a real-time enforcer were configured to use the Sentinel-generated security profiles, the enforcer could be easily designed to take a number of corrective actions in response to a detected security violation based upon the needs of the specific application.

In an audit-based implementation, Sentinel would be configured to report all security violations, while allowing the device to continue executing. This is usually the safest option to take if we do not know in advance what effects halting execution might have on a device.

In some cases halting execution might be superior to allowing execution to continue. The Sentinel device profiler can be trivially modified to cause the CPU to immediately stop executing by connecting a security status output to the reset pin of the attached CPU.

In more advanced designs, a corrective action could additionally be implemented. For example, an administrator might want to reset the device, disable network interfaces, and restore it to known-good settings so that it can continue executing in an offline-mode. For many embedded devices, Sentinel enforcers could easily be constructed to fit any of these use cases.

V. IMPLEMENTATION

In our prototype embodiment we have externally connected our Sentinel to a popular Intel 80C188-based [58] embedded device. We prototyped our bus tap on an FPGA and our security enforcer on a Linux workstation. The Sentinel bus tap is designed to be directly wired to the address pins and bus control pins of the 80C188. The bus tap observes all instruction fetches and captures the start and end addresses for each basic block that is executed. The bus tap then forwards these addresses to the security enforcer running on the Linux workstation over USB 2.0. On the workstation our device profiler prototype builds a control flow graph taken from concatenated execution traces and it enforces this profile on a captured execution trace. The device profiler algorithm runs in sub-real-time due to bandwidth constraints of our FPGA's development libraries. In a full-hardware ASIC realization this sub-real-time limitation would not exist.

A. Intel 80C188 Architecture

The Intel 80C188 is an 80186-based 16-bit x86 CPU with an 8-bit wide data bus. The 80C188XL contains 20 address pins, an address latch and three bus cycle status information pins. The bus cycle status information pins shown in Table I [58] are used to announce the type of bus operation currently in progress. The real-time bus monitor is wired directly to these bus cycle status information pins. Using this information combined with the Address Latch Enable (ALE#) pin, the bus monitor is able to decode address bus operations.

B. Bus Tap

We prototyped the Sentinel bus tap on an Opal Kelly XEM3010-1500 FPGA. This FPGA contains a USB 2.0 interface and a high-speed I/O bus which we connected to the target device through an attached breakout header. The bus tap is wired to the address and control pins and forwards all bus traffic of a chosen type over its USB 2.0 interface to a computer for analysis. The bus tap is wired to the address, bus control and address latch enable pins on the 80C188. The FPGA samples the values of these pins continuously at 133Mhz. When the address latch is active (low) and the bus control pins denote that an instruction fetch is occurring on the data bus, the bus tap saves the corresponding address from which the current instruction is being fetched. The bus tap requires that an address is valid for three sample-cycles in order to minimize the number of potential errors caused by electrical noise, crosstalk and loose wires.

If the address is stable for three cycles the bus tap checks whether the address is contiguous to the last valid address that it captured (e.g. if $currentaddress = previousaddress + 1$). If the addresses are contiguous it increments the previous address and continues to the next address. If the addresses are not contiguous then the bus tap will have captured the end of a basic block (e.g. the previous address) and the beginning of a new basic block (e.g. the next address). This implies a control flow change in the program from the previous basic block to the new basic block. The bus tap stores both of these addresses in a FIFO queue to await transfer to the device profiler.

The fact that we are not storing contiguous blocks in the FIFO is an optimization to significantly increase throughput. Rather than transmitting contiguous addresses over the USB 2.0 interface we can instead transmit block boundaries. Since the x86 architecture always begins executing at address `0xFFFF0` the receiving computer can always determine whether it is receiving the start address or the end address for a basic block. This is important for reasons that will be described in our discussion of the implementation of the device profiler.

C. Device Profiler

The Sentinel device profiler receives a list of basic block boundaries corresponding to a device execution and uses this execution trace to build a control flow graph of the associated functionality of the attached embedded system. The device profiler enforces both inter-procedural and intra-procedural control-flow constraints. Our implementation is complicated

by two issues: instruction prefetching and interrupt handling. We have solved both of these issues in our x86-specific reference implementation and our solutions are generic; they can be applied to most architectures. We have also considered architectures that utilize branch-prediction and out-of-order execution even though our 80C188 CPU did not have these features.

The Sentinel device profiler is fed a concatenated set of execution traces taken from the embedded device as it is run through its normal operations. The profiler tracks the start and end addresses for all basic blocks in each execution trace. For each address, it records the immediately preceding address as a valid possible predecessor address. That is, $(previous_block_end, current_block_start)$ and $(current_block_start, current_block_end)$ are recorded as valid jumps in the table, capturing both inter-procedural and intra-procedural control-flow. Thus the jump table forms an adjacency list representing the full control-flow graph of addresses of the instructions that fall within the security profile. Basic block boundaries are shown in more detail in Figure 1

In addition to the aggregated execution traces corresponding to the security profile, our device profiler is also fed a target execution trace on which to enforce the security profile. The device profiler checks to see if every control flow change in the target execution trace is also in the jump table corresponding to the security profile. In the event that the control-flow change is not in the security profile the Sentinel security enforcer outputs a security violation. The Sentinel security profiler will detect any unexpected control flow changes including control-flow changes caused by device errors, by a user accessing modes that are not in the security profile or by control-flow hijacking malware attacks (such as buffer overflows or many types of return-oriented programming attacks).

1) Instruction Prefetching: The Intel 80C188 CPU is capable of prefetching up to four bytes to help keep the CPU's pipeline full. This prefetching would cause problems with our security algorithms described above. In particular, the end address that we observe for a given basic block may not be the actual end of the block–it may actually be up to four bytes past the block boundary. Thus prefetching introduces a small amount of non-determinism into the system. In order to account for this nondeterminism in the system, we must allow for nondeterminism in our enforcement. Thus, to account for the prefetching we allowed for a margin of error of +/- 4 bytes in a block boundary.

In a system meant to enforce strict control-flow requirements such as a CFI-based exploit mitigation method, this margin of error could have an effect on the overall security of the system [2]. However, since the purpose of Sentinel is instead to profile and enforce mode constraints on an embedded device, this nondeterminism does not significantly weaken the underlying security of our system [3].

2) Interrupt Handling: Upon considering the security profiling algorithm above it should be clear that an obvious problem is how to account for control flow jumps generated by interrupts and exceptions.

When an interrupt is triggered on the target system the bus tap will detect a jump to the interrupt. This means that the bus tap will insert a basic block boundary immediately before the interrupt start address in the instruction stream. This boundary does not correspond to an actual block boundary in the software's basic block graph but instead is caused by the interrupt splitting the block. Similarly, an interrupt return is followed by another block boundary where the profiler begins executing again. This is illustrated in Figure 2. From an enforcement perspective, this means that jumping to or returning from an interrupt or exception should generate two consecutive violations. Thus, we can relax the restrictions on our enforcement algorithm to allow for this particular case without significantly compromising our goals. Since we are trying to disable manufacturer-implemented features and device modes rather than enforce control-flow integrity (which is an already-solved problem) we can assume that in almost all cases any useful device mode or feature that we might be interested in restricting access to would contain more than two basic blocks. Thus we can handle arbitrary interrupts without any knowledge of the underlying system and without compromising security under our threat model.

In some cases we may have access to the interrupt table (e.g. through a firmware update image). In this case we can restrict the above algorithm to only allow for jumps to known interrupts. This will make false negatives in the system even less likely than in the general case. In an x86 binary image this table is located at addresses `0x00000`—`0x00400`. From this table we can obtain the base address of all interrupt and exception handlers. We can then search the binary for the opcode of all `iret` instructions. Once we have lists of all interrupts and interrupt returns we can combine these lists to form our list of interrupt boundaries [4]. Every time we detect a control flow change we first check if it's an interrupt boundary (an interrupt start or end address). If so we skip enforcement on the previous, current and next addresses. We then continue executing normally so that we may profile the code within the interrupts. In summary, if we know the interrupt table we can restrict our algorithm to only allow for unprofiled jumps to or from known interrupt-boundary addresses.

3) Out-of-Order Execution: Many modern CPU architectures contain advanced performance optimizations such as out-of-order execution. Although the 80C188 that we tested

[2]Though this has not been established and indeed would have had to be evaluated in order to make a determination one way or the other.

[3]In order to understand why this is true consider the fact that to use prefetching-based nondeterminism to properly profile an invalid device mode as it profiles a valid device mode, the improper mode would need to be split into chunks that each fall within a maximum of n bytes of the end of a basic block where n is the maximum number of bytes that the CPU architecture can prefetch.

[4]This combination is a convenience for ease of implementation. There is no technical reason why interrupt start and end addresses could not be handled separately for slightly greater accuracy.

Bus Cycle Initiated	S2#	S1#	S0#
Interrupt Acknowledge	0	0	0
Read I/O	0	0	1
Write I/O	0	1	0
Halt	0	1	1
Instruction Fetch	1	0	0
Read Data from Memory	1	0	1
Write Data to Memory	1	1	0
Passive (no bus cycle)	1	1	1

TABLE I
BUS CYCLE STATUS INFORMATION (REPRODUCED FROM INTEL 80C188XL DATASHEET).

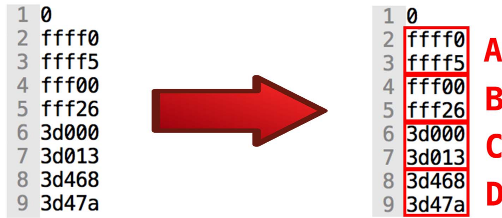

Fig. 1. Example of how a raw bus capture (left) is split into basic blocks (right).

our methodology on did not contain these features, we believe that Sentinel already supports most out-of-order execution implementations as-is. This is a corollary to the already-existing cache support included in Sentinel. In out-of-order execution, the CPU fetches instructions in program order and stores them in a cache [60]. The CPU then pulls instructions from the cache in an optimized order. Thus the out-of-order step occurs after the instruction fetch from memory. Since Sentinel concerns itself only with instruction fetches that occur outside of the CPU, out-of-order execution should have no impact on Sentinel at all.

VI. EVALUATION

We evaluate the Sentinel architecture by proving its real-world efficacy in detecting a physical attack on an embedded medical device. The process for this evaluation is as follows:

1) Create security profile
2) Enforce security profile
3) Access mode not defined in profile
4) Detect attack in execution trace

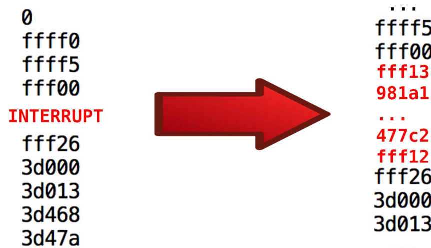

Fig. 2. Example of how interrupts split a basic block.

A. Alaris SE Infusion Pump

We evaluate the Sentinel architecture by connecting our working prototype to a popular embedded medical device. For our test implementation, we have selected the Alaris Signature Edition (SE) Infusion Pump due to its popularity and its x86-based architecture. We looked at two different board layouts for the Alaris SE. The Alaris SE contains an Intel 80C188-based [58] processor (or, in some revisions, the AMD equivalent), a custom ASIC for address encoding/decoding and data bus multiplexing, and either an Intel E28F800-CVT70 [61] flash memory or an STM M27C801 UV EPROM. In board revisions that contain the Intel E28F800 the manufacturer added a header to emulate the pin layout of the STM M27C801 as shown in Figure 3. When describing information that is pertinent to both chips we will subsequently use the term ROM. The STM M27C801-based pumps contain a socketed EPROM that can be read using a standard universal flash programmer. We dumped the firmware from the STM M27C801 by using a flash programmer to receive a binary image of firmware version 2.79.

The Alaris SE logic board also contains a 128K battery-backed SRAM to which some code is copied at boot and in which persistent settings are stored. The Intel 80C188 is capable of addressing up to 1MB of memory. The onboard ROM is 1MB and thus occupies the entire address space. The SRAM overlays the memory region beginning with address `0x3D000`. Upon system boot the device first copies 6144B of data from `0xFD000` to `0x3D000` before jumping to `0x3D000` where it begins executing.

B. Connecting the Bus Tap

We connected our bus tap to the infusion pump by soldering wires to each address pin, the CPU bus control pins (S_0, S_1 and S_2) and the ALE# pin. Our particular infusion pump selectively enabled the ROM or the SRAM through the use of chip enable pins wired from the ASIC to the respective chips. The address pins connected to both the ROM and the SRAM through the ASIC and used in conjunction with the chip enable pin could be used to send an address to either the ROM or the SRAM. The address pins wired to both of these chips always showed the same addresses as the address pins on the CPU. Thus to simplify the soldering we actually soldered our wires to the address pins connected to the ROM rather than to the CPU as shown in Figure 4. Note that this shortcut was discovered by our testing of the Alaris SE and is thus not necessarily generalizable to other devices. We would not recommend a similar shortcut in a real-life deployment without first performing similarly rigorous testing.

C. Error Correction

Due to the requirement that the bus tap interface only send valid addresses, the bus tap will occasionally drop an address if an error occurs. When this happens in the middle of a block,

the bus tap will see two noncontiguous addresses and think that a jump has occurred. If it happens at the beginning of a block the bus tap will start the block one address after the actual starting address of the block. If it happens at the end of a block the bus tap will end the block one address before the actual ending address of the block (which may or may not be the real ending address because prefetching makes this impossible to discern). We have trivially handled each of these errors in software.

In our test implementation, we did not implement reliable delivery over our bus tap interface due to overhead in our FPGA libraries. Even without reliable delivery we were able to reliably build and enforce profiles. The lack of reliable delivery affected profile building but it had no impact on profile enforcement. Indeed, in our tests, different device modes looked so different from the perspective of our enforcer that it was easy to distinguish between them. Similarly, despite the lossy interface we were able to build enforceable profiles with low false positive rates with just one sample, and we were able to build enforceable profiles with no false positives at all by the time we reached three samples. Thus, by profiling a device during typical use over a short period of time we can build reliable and enforceable profiles even with respect to lossy interfaces. This means that in practice a bus monitor can be significantly weaker than the system it is attached to, thus lowering the overall cost.

D. Enforcing a Profile

To demonstrate the efficacy of the Sentinel platform we use it to build a profile of a normal device booting and navigating to the ”Infuse” screen where a healthcare provider can configure infusion rate and amount and start an infusion. We intentionally build the profile to not include the ”Options” screen. Thus, if a user accesses the options screen this should be flagged as a security violation. We show that we are able to build an enforceable profile with no false positives after just three samples.

1) Testing the Profile: In order to test the efficacy of enforcing a security profile we took eight execution traces each containing 8,388,608 addresses. The execution traces consisted of the addresses of all basic block boundaries observed by our bus monitor while we booted the pump to the ”Infuse” screen. We also took a ninth execution trace of the device booting to the ”Infuse” screen to use as an experimental control. Our test case consisted of a capture of the first 8,388,608 basic block boundary addresses observed by our bus monitor while we booted the pump to the ”Infuse” screen and then navigated to the ”Options” menu. Note that in all cases, parts of the captured data were null to some screens were null due to data loss through our system. That is, our captures were lossy.

We created eight profiles to enforce by concatenating up to eight execution traces together. For example, profile one contained execution trace one, profile two contained execution traces one and two, profile three contained execution traces 1-3 and so on. We enforced each profile on our control sample and on our experimental sample. In our control sample we would expect to see no security exceptions since we are enforcing our ”Infuse” profile on an ”Infuse” test case. In the experimental sample we would expect to see numerous security exceptions since we are enforcing the ”Infuse” profile while the user accesses an unprofiled device mode. The results of our experiment confirm our hypotheses and are reflected in Table II.

2) Discussion: From our results we see that our method of capturing lossy partial execution traces can be used to reliably differentiate between device modes and to enforce profiles. We see that in the control test we had a few initial false positives but that the number of false positives quickly decreased to zero as we added more samples to our profile.

We also observed that we were immediately able to differentiate between our control and experimental test cases even with just one experimental sample in our profile. As we concatenated more samples to build our profile the number of false positives quickly dropped. After combining four samples into a profile all false positives not caused by physical errors had been eliminated. Thus, our results imply that only a few samples are needed in order to successfully profile a given device feature. Furthermore, accessing a ”locked” device mode generated several orders of magnitude more security errors than accessing a profiled device mode and thus even if we had not combined multiple samples to build a device profile we still could have distinguished between the control and experimental cases with high accuracy.

3) Automation Through Keypad Emulation: Because we have shown that building accurate profiles is best accomplished by combining multiple samples, we devised a method for automatically generating samples by emulating keypad inputs on our Alaris SE infusion pumps.

In order to differentiate key presses, the host device typically uses a microcontroller to poll the button rows in sequence. When a key is pressed, the microcontroller reads from the column output and derives the button location in the matrix. When no keys are pressed, there is no output on the column lines. When a key is pressed, one of the column lines will be logically active and the microcontroller can determine the individual button based on the row it was polling.

The Alaris SE uses 24 buttons. The row and column lines are fed to the host device through two 8-pin FFC (Flexible Flat Cable) cables. In order to access these lines, we soldered hookup wires to the the mount points on the FFC connectors on the underside of the PCB. Using a logic analyzer, we reverse-engineered the electrical signals corresponding to various keypresses. Since buttons in the same column trigger output on the same line, the first step was to map out the complete button layout to determine which pins corresponded to column lines. The row lines were trivial to ascertain with the logic analyzer since the microcontroller polls were clearly evident. We found the column lines by pressing each button and observing which line responded. Using this process we discovered that there were five row lines and seven column lines. The remaining lines were unused.

Once each button’s column was known its corresponding

# of Samples in Profile	1	2	3	4	5	6	7	8
# Errors in Control Sample	2	2	0	0	0	0	0	0
# Errors in Experimental Sample	9250	9220	9049	8705	8374	8367	8365	8358

TABLE II
RELATIONSHIP BETWEEN NUMBER OF SAMPLES IN PROFILE, FALSE POSITIVES, AND DETECTION ACCURACY.

Fig. 3. Board Layout of the Alaris SE 7132. Redundant SRAM chips on left, CPU top-middle, flash between header pins, ASIC top-right.

row was discovered by comparing the output signal to one of the five possible inputs. Since holding down a button will result in the exact signal output as one of the inputs, this process was simply done through direct comparison and was exhaustively applied to determine the matrix location of each button. Once complete, it was possible to wire a given row to a column and force the microcontroller to interpret a button press. This enabled external command of the keypad interface without physically pressing buttons.

Using GPIO on a Raspberry Pi we replaced desirable buttons by wiring their corresponding row and column lines through a relay. We used a script to activate buttons in sequence, allowing for total control over the device interface.

VII. Future Work

Some advanced architectures utilize branch prediction and branch target prediction in order to improve performance. Although Sentinel has not been tested on such architectures, we believe that it should support them with extremely minor modifications. Because Sentinel, in its current implementation, allows two consecutive invalid jumps before it flags a mode as suspicious, a missed branch prediction should not trigger an alert. However, we have not tested this and it is conceivable an alert may be triggered if an interrupt occurs after the CPU has fetched instructions for a predicted branch that isn't in the profile that Sentinel is enforcing. This could be remedied by allowing for one additional level of indirection before triggering an alert (e.g. by allowing three consecutive invalid jumps before triggering an alert).

Because the 80C188XL found in our test device does not support features such as branch prediction or out-of-order execution we were not able to test our out-of-order execution algorithm in practice. We believe that this would be a useful exercise as it would improve confidence in our algorithm and definitively open up Sentinel to a new class of devices.

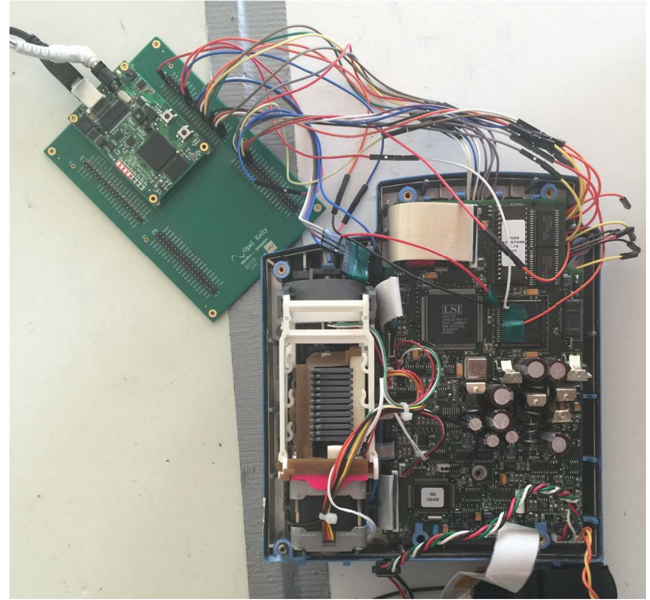

Fig. 4. Image of Sentinel Bus Tap attached to an Alaris SE 7100 Infusion Pump.

It would also be interesting to conduct an observational study in order to determine how often devices deviated from their execution profiles in a variety of use cases. This would allow us to derive an understanding of applications for which Sentinel would be well-suited and for which applications Sentinel would be poorly suited given that Sentinel's enforcement model depends on having a predictable and inclusive execution profile.

VIII. Conclusion

Sentinel is a useful and practical tool for utilizing bus captured execution traces to build partial control flow graphs of embedded devices. These control flow graphs can be used to audit execution traces of embedded devices in order to detect physical attacks. The Sentinel platform can be attached to existing devices with no modifications to the underlying design. We have shown our bus monitoring technique to be effective in building partial control flow graphs and we have used it to successfully detect physical attacks on a popular legacy model of infusion pump. In the future, this work could lead to ASIC-implemented hardware co-processors that can be used to secure later revisions of legacy designs by enforcing limited-use device profiles on otherwise feature-rich devices in real-time.

ACKNOWLEDGMENT

The authors would like to thank Gary Truslow, Joseph Carrigan, Katie Chang and Yifan Yu for their contributions to the design and testing of Sentinel.

REFERENCES

[1] Symantec, "Solution overview: Industry perspectives - healthcare embedded medical device security and privacy," dec 2010.

[2] Lantronix, "Medical device networking for smarter healthcare - real world integration ? applications, integration issues and benefits," dec 2010.

[3] S. R. Rakitin, "Networked medical devices: Essential collaboration for improved safety," *MANAGEMENT & TECHNOLOGY*, pp. 332–338, jul 2009.

[4] H. Baldus, K. Klabunde, and G. Muesch, "Reliable set-up of medical body-sensor networks," in *Wireless Sensor Networks*. Springer, 2004, pp. 353–363.

[5] R. Dantu, H. Oosterwijk, P. Kolan, and H. Husna, "Securing medical networks," *Network Security*, vol. 2007, no. 6, pp. 13–16, 2007.

[6] C. Doukas, I. Maglogiannis, and G. Kormentzas, "Advanced telemedicine services through context-aware medical networks," in *International IEEE EMBS special topic conference on information technology applications in biomedicine*, 2006.

[7] C. Doukas and I. Maglogiannis, "Adaptive transmission of medical image and video using scalable coding and context-aware wireless medical networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2008, p. 25, 2008.

[8] T. Gao, C. Pesto, L. Selavo, Y. Chen, J. Ko, J. H. Lim, A. Terzis, A. Watt, J. Jeng, B.-r. Chen *et al.*, "Wireless medical sensor networks in emergency response: Implementation and pilot results," in *Technologies for Homeland Security, 2008 IEEE Conference on*. IEEE, 2008, pp. 187–192.

[9] Midas, "Midas technical handbook." [Online]. Available: http://microwatt.com/wp-content/uploads/2014/10/MIDASA001-Technical-Manual-ENG-rev18.pdf

[10] "Micrologix 1100 embedded web server." [Online]. Available: http://literature.rockwellautomation.com/idc/groups/literature/documents/um/1763-um002_-en-p.pdf

[11] "Medical devices contain hard-coded passwords, ICS-CERT warns." [Online]. Available: http://threatpost.com/medical-devices-contain-hard-coded-passwords-ics-cert-warns/100994

[12] ICS-CERT, "Icsa-15-309-02." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-15-309-02

[13] ——, "Icsa-15-300-03." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-15-300-03

[14] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *IEEE Symposium on Security and Privacy, 2008. SP 2008*, May 2008, pp. 129–142.

[15] C. Li, A. Raghunathan, and N. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *2011 13th IEEE International Conference on e-Health Networking Applications and Services (Healthcom)*, Jun. 2011, pp. 150–156.

[16] Alaris, "Alaris medical systems ivac model 710x 720x series signature edition volumetric pump technical service manual," 1997.

[17] D. M. Zuckerman, P. Brown, and S. E. Nissen, "Medical device recalls and the fda approval process," *Archives of internal medicine*, vol. 171, no. 11, pp. 1006–1011, 2011.

[18] E. Snell, "Fda releases medical device cybersecurity warning," 2015.

[19] D. Arora, S. Ravi, A. Raghunathan, and N. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 12, pp. 1295–1308, Dec 2006.

[20] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 511–526. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534810

[21] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, pp. 144–148.

[22] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 461–491, 2004.

[23] Z. Shao, Q. Zhuge, Y. He, and E.-M. Sha, "Defending embedded systems against buffer overflow via hardware/software," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. IEEE, 2003, pp. 352–361.

[24] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao, and E.-M. Sha, "Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software," *Computers, IEEE Transactions on*, vol. 55, no. 4, pp. 443–453, 2006.

[25] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the first ACM workshop on Secure execution of untrusted code*. ACM, 2009, pp. 19–26.

[26] R. Riley, X. Jiang, and D. Xu, "An architectural approach to preventing code injection attacks," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 351–365, 2010.

[27] S. Lukovic, P. Pezzino, and L. Fiorin, "Stack protection unit as a step towards securing mpsocs," in *Parallel & Distributed Processing, Workshops and Phd Forum*

(IPDPSW), 2010 IEEE International Symposium on. IEEE, 2010, pp. 1–4.
[28] F. Wolff, C. Papachristou, D. Weyer, and W. Clay, "Embedded system protection from software corruption," in *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*. IEEE, 2010, pp. 223–229.
[29] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.
[30] J. Radcliffe, "Hacking medical devices for fun and insulin: Breaking the human scada system," in *Black Hat Conference presentation slides*, vol. 2011, 2011.
[31] D. Weber, "Optiguard: A smart meter assessment toolkit," *Black Hat USA*, 2012.
[32] ICS-CERT, "Icsa-10-214-01." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-10-214-01
[33] ——, "Icsa-15-300-02." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-15-300-02
[34] ——, "Icsa-11-216-01." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-11-216-01
[35] ——, "Icsa-12-030-01a." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-12-030-01A
[36] ——, "Icsa-12-018-01b." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-12-018-01B
[37] Alaris, "Alaris 8015 infusion pump."
[38] Mitsubishi, "Mitsubishi plc fx2n."
[39] ——, "Mitsubishi programmable controller fx3g-40mr/es-a."
[40] Allen-Bradley, "Allen-bradley plc-5 1785-me32."
[41] GE, "Ge eagle3000 patient monitor."
[42] Diebold, "Diebold opteva atm machine depositor pcb 49-2000330-000e."
[43] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251388
[44] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 28–37. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382202
[45] K. Flanagan, "Bach: Byu address collection hardware, the collection of complete traces," 1992.
[46] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 392–403. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485956
[47] F. Vahid and T. Givargis, *Embedded system design: a unified hardware/software introduction*. John Wiley & Sons New York, NY, 2002, vol. 4.
[48] W. Stallings, *Computer organization and architecture: designing for performance*. Pearson Education India, 2000.
[49] P. M. Kogge, *The architecture of pipelined computers*. CRC Press, 1981.
[50] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981, pp. 135–148.
[51] K. Driesen and U. Hölzle, "The cascaded predictor: Economical and adaptive branch target prediction," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 1998, pp. 249–258.
[52] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*. Newnes, 2013.
[53] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.
[54] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.
[55] J. J. Carr, *Z80 User's Manual*. Prentice Hall PTR, 1980.
[56] I. Freescale Semiconductor. M68000 8-/16-/32-bit microprocessors user?s manual. Freescale Semiconductor, Inc. [Online]. Available: http://www.freescale.com/files/32bit/doc/ref_manual/MC68000UM.pdf
[57] D. Seal, *ARM architecture reference manual*. Pearson Education, 2001.
[58] Intel, "80c186xl/80c188xl 16-bit high-integration embedded processors," oct 1995.
[59] FDA, "Sigma spectrum infusion pumps with master drug library by baxter healthcare: Class i recall - system error may interrupt or delay therapy." [Online]. Available: http://www.fda.gov/Safety/MedWatch/SafetyInformation/SafetyAlertsforHumanMedicalProducts/ucm395770.htm
[60] J. U. Skakkebæk, R. B. Jones, and D. L. Dill, "Formal verification of out-of-order execution using incremental flushing," in *Computer Aided Verification*. Springer, 1998, pp. 98–109.
[61] Intel, "8-mbit (512k x 16, 1024k x 8) smartvoltage boot block flash memory family," sept 1995.