Hardware Based Detection, Recovery, and Tamper Evident Concept to Protect from Control Flow Violations in Embedded Processing

Patrick R. DaSilva
Naval Undersea Warfare Center Division Newport
Newport, RI, U.S.
patrick.dasilva@navy.mil

Paul J. Fortier
Electrical and Computer Engineering Department
University of Massachusetts Dartmouth
North Dartmouth, MA, U.S.
pfortier@umassd.edu

***Abstract*—The ubiquitous presence of embedded devices coupled with their low processing power and finite energy creates unique challenges in the security of embedded systems. Software attacks targeted at maliciously modifying the control flow of an executing embedded program negatively affect real-time response. Hardware-based control flow violation detectors have been researched and tested, but still lack the means to recover from control flow attacks and provide meaningful data for post cyber-incident analysis.**

Proposed is a hardware-based system on a chip (SoC) concept to protect low-end embedded processors from control flow attacks. The concept provides an end-to-end protection combination of detection, response, recovery, and tamper evident techniques against control flow violations in the presence of interrupts, real-time operating systems, and exceptional functions. The implemented detection portion of the solution shows promise to detect most CRAs on low-end embedded systems without added cycle latency at the cost of increased area.

Keywords—control flow integrity, tamper evidence, sacrificial core, detection, response, recovery, embedded systems security

I. Introduction

Attacks on embedded systems aren't new but are common thanks to the rise of the Internet of Things (IoT) and improvements in embedded processor technology. To date, there is no single source that categorizes attacks on embedded systems but lumps them into cybersecurity incidents [1]. Given current trends in cybersecurity, global organizations are not fully prepared to handle sophisticated cyber-attacks, let alone attacks on embedded systems. According to Prevelakis et al. [2] worldwide critical infrastructures such as Energy, Finance, Food, Water, and Health are vulnerable to embedded system attacks and have all been affected by cybersecurity incidents.

An embedded system's program control is commonly the target of cyber-attacks. Control flow violations in various embedded components can lead to arbitrary code execution, unauthorized control, and denial of service [3]. Most control flow vulnerabilities can be exploited remotely and don't require any specialized knowledge or skill [3]. Malicious attempts to change an embedded control flow entail program memory injection, data memory execution, program counter manipulation, or direct injection into the pipeline as in the case of Hardware Trojans. A control flow attack can leave a real-time system in a non-functioning state, affecting the application and environment.

Low-end embedded systems require the ability to detect, respond, recover, and collect tamper evident information on control flow attacks. Given their limited core processing power, available energy, physical exposure, and network connectivity, embedded devices need an integrated hardware-based approach to provide a complete real-time security solution while operating in an unsupervised environment.

Proposed is a concept to implement detection, response, recovery, and tamper evident hardware circuitry alongside an embedded soft-core processor while studying trade-offs to obtain different levels of protection. The target embedded processor, a low-end Alf and Vegard RISC (AVR) soft-core, and security solution are implemented on a Xilinx Field Programmable Gate Array (FPGA). The objective is to protect the AVR soft-core from exploitable vulnerabilities by detecting control flow violations, providing appropriate recovery methods, and gathering evidence of the attack to avoid future harm to the embedded system environment.

Section II will elaborate the background needed to understand topic significance and hardware concept. Section III will describe the proposed hardware concept and section IV will present preliminary results. Finally, section V will summarize the paper as well provide future work.

II. Background

A. Embedded Systems Security

Embedded Systems Security (ESS) stems from Computer Security as it applies to embedded systems and its assets. ESS reduces vulnerabilities and provides protection against threats to embedded system design, hardware, firmware, system software, data, communications, and networks. Each of these seven assets are a security domain in themselves and together form ESS. With trends such as IoT and remotely-controlled industrial

This work was supported by the Naval Undersea Warfare Center, Section 219 Research Program, Anthony Ruffa, program manager.

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

systems, embedded systems became less obscure and more convenient to attack.

To aid in fortifying computer security from growing computer system vulnerabilities, the Framework for Improving Critical Infrastructure Cybersecurity [4] was released in April 2018. The framework is provided to critical infrastructure owners and operators for voluntary use to address the effect of cybersecurity on physical, cyber, and people. The framework core consists of five functions or basic cybersecurity activities; Identify, Protect, Detect, Respond, and Recover. The National Institute of Standards and Technology (NIST) developed framework can be applied to embedded systems security to manage the cybersecurity risk posed to critical infrastructures. If a threat to a vulnerability in an embedded device is identified, protected from, detected, responded to, and recovered from, then the cybersecurity risk to the critical infrastructure is managed.

B. *Control Flow Attacks on Embedded Systems*

Control flow attacks encompass code reuse attacks (CRAs) and code injection attacks. Using a control flow graph (CFG), the steps through a process can be visualized as a set of directed edges and nodes. An unaltered process will follow the correct forward and backward edges. Any redirection of an edge would cause a control flow violation, Fig. 1.

Software-based control flow attacks exploit vulnerable sub-routines to alter data on the stack or heap of a processor. Vulnerable data that modify control flow include function return addresses, function pointers, and setjmp/longjmp data buffers stored on the stack.

CRAs assume the attacker has no control over program memory but has enough knowledge about it to create gadgets using exploits like buffer overflow. A gadget is a sequence of instructions which ends in an indirect branch instruction to a target controlled by the attacker. In return-oriented programming (ROP), jump-oriented programming (JOP), and call-oriented programming (COP), a gadget ends in a return, jump, or call instruction respectively. Each gadget is constructed from existing words in program memory to perform a small task

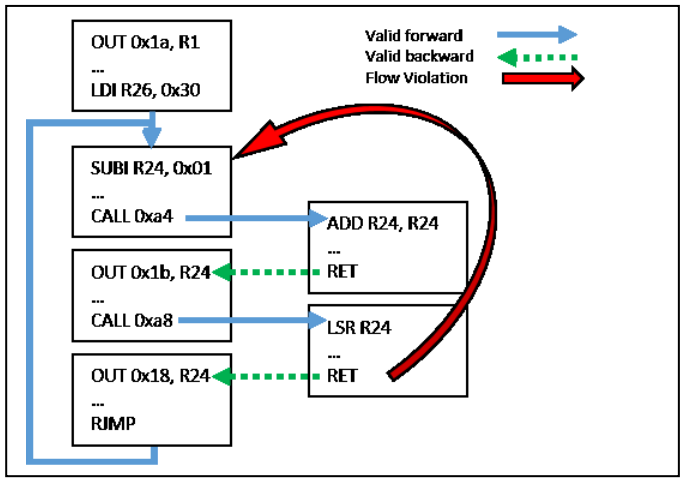

Fig. 1. Example control flow graph with violation.

like loading a register or placing contents into memory and then setting up the next gadget. Gadgets are strung together to make a Turing complete attack linked through return, jump, or call instructions.

Code injection attacks inject instructions directly into program memory (PM). On the Von Neumann architecture, data and program memory share the same bus and memory space, making Von Neumann machines vulnerable to code injection. Alternatively, the Harvard Architecture contains separate data and program memory areas enforcing the PM to be read-only and data memory (DM) to be only used for data. This was true until instruction set architectures (ISAs), like AVR, provided instructions to load program memory from data memory allowing for data as instructions e.g. boot loaders. Boot loader instructions allow for code injection attacks to be possible on the AVR ISA by first launching a CRA through a buffer overflow exploit [5].

C. *Hardware-Based Control Flow Integrity*

A hardware-based control flow integrity (CFI) solution is required to reduce the impact on compute time to detect control flow violations. CFI focuses on maintaining the integrity of a CFG, constructed from analyzing a program's binary, using backward and/or forward edge policies.

Backward edge policies focus on protecting a program's control flow from CRAs that corrupt the return address of a function (e.g. ROP). There are various ways to implement a backward edge policy, but the most popular and recommended as necessary in [6] is to use a shadow call stack (SCS). An SCS is an additional stack implemented in hardware to hold the return address of a function call. Upon return, the hardware checks the return address on the call stack with the one on the SCS. If they're not the same, a potential integrity violation ensued. Challenges of a hardware-based SCS include, optimizing the hardware buffer size and unrolling the SCS after exceptions like longjmp and setjmp [7][8][9].

Forward edge policies account for direct and indirect addressing. Direct addressing (e.g. call, jump, branch, skip) can be obtained from the CFG and instruction itself and are therefore not as vulnerable to control flow violations, so long as a CFG is maintained. Indirect addressing (e.g. IJMP, ICALL, RET) are vulnerable and therefore harder to detect a control flow violation within. Rules are then used to constrain the potential target address of these instructions.

Das et al. [9] developed BB-CFI which limits branch targets to basic block boundaries. BB-CFI uses a modified set of rules to combat exceptions that may arise; (1) CALL can target the basic block of a function, (2) RET can only target the basic block address of the call site or a basic block for an exception handler, (3) IJMP can target a return address or the starting address of a basic block, and (4) RET can target any address in the return address stack. Rule 2 is directed at C++ exception handling and rule 4 applies to multi-threading. Rule 3 addresses longjmp and setjmp functions.

Like most CFI policies [10][11], BB-CFI is integrated into the pipeline stages of the ISA and requires metadata for its basic

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

block control graph from the program's CFG. BB-CFI includes an SCS, basic block table, control unit, and buffer containing control flow instructions that still need to be validated. Detection accuracy suffers if the CFG is not complete.

Nyman et al. [12] proposed CaRE, a hardware-supported CFI solution that is interrupt aware for commercial microcontrollers. Low-end microcontroller applications, such as IoT, are often interrupt driven to maintain strict timing requirements. Most hardware-based CFI solutions [8][9][13] fail to mention rules that are interrupt aware.

Similar to HAFIX [13], CaRE [12] adds instrumentation into the program binary trading off increased program binary size and execution time for CFI violation detection.

D. Response, Recovery, and Tamper Evidence

Current hardware-based CFI solutions barely mention any response or recovery details other than providing a signal for external response [11] or rebooting the entire processor [14]. Mao et al. [10] and Thomas [15] do mention response and recovery, but both require operating system support.

In a real-time environment, recovery time is crucial to allow the embedded system to resume its functions as quick as possible. Ferguson and Gu [16] describe a sensor self-healing option implemented on an ATmega128 by embedding diversified protection and recovery code at particular locations to enforce access control in code memory. The software-based solution presented in [16] leads to question the ability to use a hardware-based solution to perform a backward error recovery through checkpoints [17][18][19]. Backward error recovery techniques are based on time redundancy and rely on saving the correct processor state. To reduce the area and time overhead, careful decisions need to be made on how much of the processor context needs to be saved and how often.

Reducing the impact of recovery can be done by isolating the attack and allowing the system to run in a degraded state as outlined by Thomas and Abdelwahed [20]. To allow the mission critical system to run in a degraded state, the authors proposed to route mischievous functionality from a malicious process to a sacrificial virtual machine. Embedded systems don't normally have the resources to run a virtual machine but diverting malicious functionality to a sacrificial soft-core on an FPGA could reduce the negative impact of a full recovery, if one is not required.

III. HARDWARE CONCEPT

The hardware concept augments an AVR soft-core with a security solution on a Xilinx FPGA. The security solution, depicted in Fig. 2, contains detection, response, recovery, and tamper evident activities to protect from exploitable vulnerabilities by detecting control flow violations, providing appropriate recovery methods, and gathering evidence of the attack to avoid future harm to the embedded system environment. Solution objectives include (1) minimize or avoid detection latency, (2) minimize system recovery latency, (3) minimize embedded system impact of detected activity, (4) capture and provide tamper evident data, (5) avoid or minimize modifications to existing binaries, (6) avoid altering AVR instruction set, and (7) avoid re-synthesis of hardware solution to accommodate new program binaries.

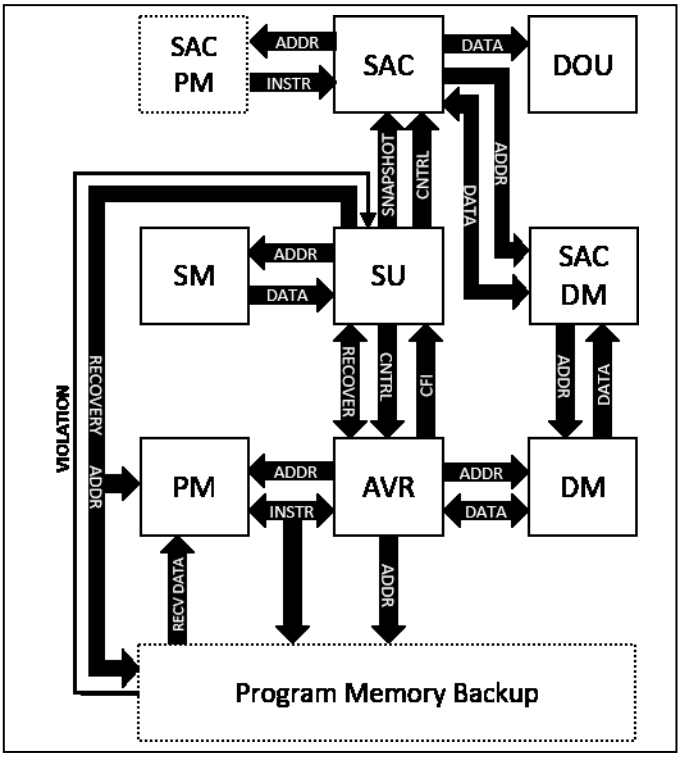

Fig. 2. Block diagram of general protection solution.

The three main subsystems, target, security, and tamper evidence are centered around the AVR soft-core, security unit (SU), and sacrificial AVR soft-core (SAC), respectfully. Detection, response, and recovery are overseen by the SU which also initiates the tamper evident subsystem. The AVR soft-core is connected to the program (PM) and data memory (DM) as normal but is augmented to provide necessary CFI information to aid CFI violation detection and snapshots of its present state to aid the recovery process.

A. Offline Metadata Collection and Boot-Up Process

To aid in detection of control flow violations, metadata from an off-line control flow analysis conducted on the program binary is loaded into the security memory (SM) through a merging process with the FPGA bit stream file. This avoids the need to re-synthesize the FPGA bit stream for each new binary.

Metadata collection includes a static-binary analysis to construct a control flow graph and provide basic block, Real-Time Operating System (RTOS), and setjmp/longjmp usage information. Information collected during this process includes:

- Basic Block beginning and ending address and a flag set if it is the first block of a function
- RTOS start address of the save and restore context routines, data to prime each task stack, memory location of task identifier, and default operating system shadow call stack pointer value

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

- Start address of setjmp and address of IJMP instruction within longjmp
- Allowed serviceable interrupt vectors
- ICALL instruction addresses with corresponding allowed targets

The aassumption is this information is available by inspection of the Executable and Linkable Format (ELF) file and CFG by an individual with knowledge of how the real-time embedded system should behave.

On boot-up, the metadata is loaded into the SU CFI memory (CFI MEM), Basic Block (BB) Content Addressable Memory (CAM), and Shadow Call Stack (SCS) from the SM and used during the detection process, depicted in Fig. 3.

B. Detection

As stated in [5], code injection attacks on the AVR platform are possible through CRAs. The detection portion for this system, therefore, focuses on detecting control flow violations through the use of code reuse attacks, specifically those that target call, jump, and return instructions. The detection system, shown in Fig. 3, is similar to other CFI solutions in that it makes use of a basic block table, shadow call stack, and control unit, but other restrictions, exceptions, and improvements were made. Table I summarizes the basic rules used in most CFI solutions [8][9].

TABLE I. BASIC CFI RULES

Basic Rules
1. ICALL targets function entry
2. IJMP targets a basic block
3. RET targets the address following the corresponding CALL
4. CALL-RET pairing is enforced

The basic CFI rules listed in Table I cover the majority of normal control flow programs, but there are exceptional cases that break the normal control flow. The sections below describe the complete detection ruleset implemented in our solution and calls out what was an addition, exception, or restriction.

1) ICALL target addresses (TAs) must target the first basic block of a function. Upon ICALL execution, the TA is provided to the BB CAM which provides an address into the basic block table resident in the SM. The target basic block information then provides a flag as to whether the target is function block or not. A restriction on this rule is if the basic block containing the ICALL instruction has specified Allowed Indirect Target Addresses (AITAs), then the ICALL instruction must target one of the specified targets. BB REG within Fig. 3 holds information on the current executing basic block in order to provide the AITAs. BB REG is updated based on the current executing control transfer instruction.

2) IJMP target addresses must target a basic block entry. Upon IJMP execution, the TA is provided to the BB CAM which provides a signal to determine if the address is a basic block or not. IJMP instructions within setjmp and longjmp routines do not set off a violation as they target basic block entries, but subsequent RET instructions cause false positive violations because the stack is unwound and the SCS is not updated during longjmp executions. To restrict possible IJMP targets of a setjmp buffer violation, the detection solution is implemented with a SetJmp (SJ) CAM which is updated with the current TA, Stack Pointer (SP), Task Identifier (TID), and Shadow Call Stack Pointer (SCSP) on each call to setjmp. A restriction on the original rule is an IJMP instruction within a longjmp function must target the corresponding setjmp buffer for a TA/SP/TID pair. If found, the SCSP is provided to unroll the SCS to when the setjmp function was called.

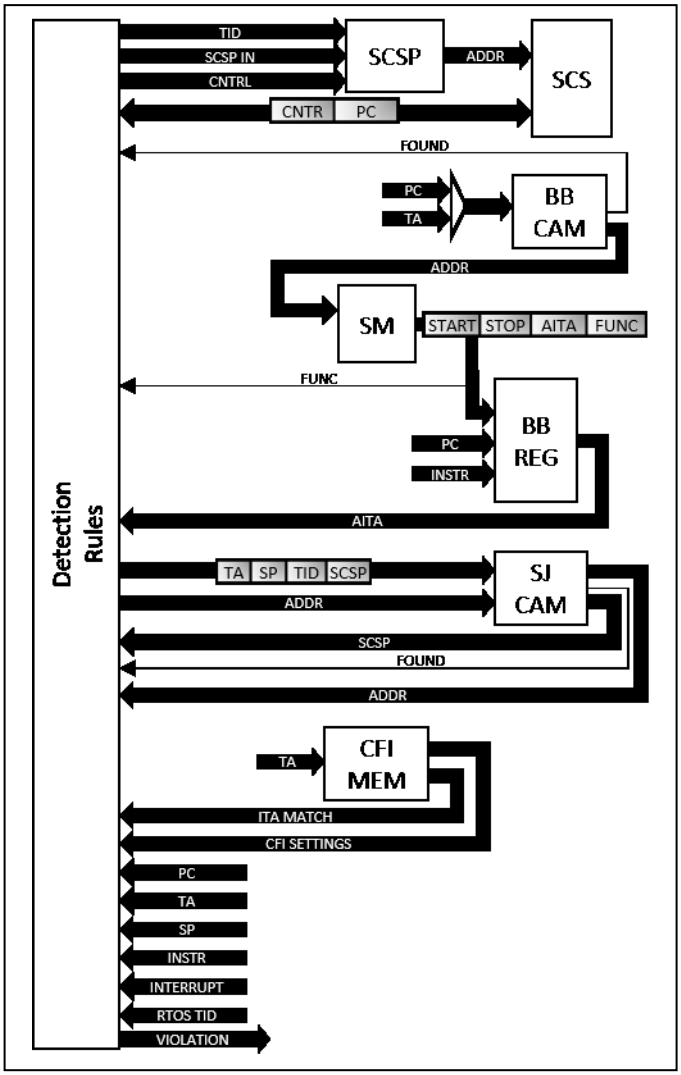

Fig. 3. General detection solution.

3) RET and RETI instructions must always target the top of the SCS. To support the RETI instruction, an addition was to monitor interrupts and place the Program Counter (PC) onto the SCS each time an interrupt occurred.

4) CALL-RET pairing is enforced. The address following a CALL/RCALL/ICALL is put onto the SCS to enforce CALL-RET pairing. The first exception to this rule is any

CALL/RCALL to setjmp. The function setjmp will always remove the PC from the stack and then invoke an IJMP instruction. The second exception is any CALL/RCALL/ICALL to a target immediately following the instruction. This is primarily seen in the form of RCALL+0 instructions when room needs to be made on the stack for a short buffer. An addition to the orginal rule is when setjmp is called, the SJ CAM is updated with the current PC/SP/TID/SCSP to support subsequent calls to longjmp, as described in rule 2. Recursive function calls are supported with the use of a recursive counter appended onto each entry within the SCS, similar to the work described in HAFIX [13]. This reduces the required size of a SCS.

5) (Addition) Servicable Interrupts are restricted to specific allowed interrupt vectors. Upon an interrupt, the interrupt vector address is checked against the allowed interrupt vectors identified in CFI MEM.

6) (Addition) Hardware-Based SCS is RTOS aware through the CFI MEM loaded with the metadata from the offline analysis. Rules 1 through 5 are supported through the current active SCS, where applicable. The SCS is divided into 17 stacks to support 16 tasks and 1 Operating System (OS) stack. Active SCS is maintened by control flow transfers to special OS functions called saveContext and restoreContext. Most RTOS implementations will have these functions. A CALL/RCALL to saveContext changes the TID input into the SCSP register to the OS and saves the current PC on the OS SCS at the default OS SCSP while a JMP/RJMP to saveContext just changes the TID. A CALL/RCALL to restoreContext changes the TID input into the SCSP register to the current RTOS TID and saves the current PC onto the task SCS while JMP/RJMP just changes the TID to the RTOS TID.

C. Response and Recovery

Before each valid CALL instruction, a snapshot of the core is taken. Upon CRA detection, a previous valid snapshot is loaded into the AVR core and the core continues processing from a state before the attack happened. If the same CRA is encountered, then a reset occurs to clear the DM. If a code injection attack is detected, partial or whole portions of the program memory will be reloaded from the program memory backup (PMB) to recover the system to a usable state. As an added benefit, code integrity can be checked against the PMB as each instruction word leaves the PM.

D. Tamper Evidence

The default response to a detection is to isolate impact of the malicious software from the rest of the application using a similar idea proposed by Thomas and Abdelwahed [20]. The tamper evident system is composed of a sacrificial AVR soft-core loaded with checkpoint data taken at time of detection prior to the AVR soft-core recovery. The SAC objective is to collect information on the malicious process for post cyber-incident analysis while allowing the embedded core to continue operating with a reduced negative impact. If anything can be learned as to why this attack happened, the binary can be fixed on future updates.

While the AVR core is recovered, the SAC is enabled and allowed to run. For detected CRAs, the SAC has read access to the same program memory. To maintain read-read and write-read data properties while reducing area overhead and recovery latency, all reads and writes from the SAC to the DM will be through the SAC DM. The SAC DM uses a page table to map reads and writes to the data memory. To maintain the illusion that the malicious process is running undetected, the sacrificial core's IO will be spoofed. The SAC's decoded instructions and program counter will be stored on external storage through the data output unit (DOU) to aid post-cyber incident analysis.

Recovered code injection attacks overwrite the PM and no longer allow the SAC to process from the same PM as the AVR core. The SAC PM will be loaded with the contents of the AVR's PM to preserve the function of the SAC.

IV. Preliminary Results and Disscussion

Implementation thus far has included the offline metadata collection and CFI detection system with hooks for response and recovery as well as tamper evidence. Table II shows how well the hardware concept has met the requirements thus far.

TABLE II. Hardware Concept Requirements

Requirement	Result
1. Minimize or avoid detection latency	Avoided
2. Minimize system recovery latency	Pending
3. Minimize embedded system impact of detected activity	Pending
4. Capture and provide tamper evident data	Pending
5. Avoid or minimize modifications to existing binaries	Avoided
6. Avoid altering AVR instruction set	Avoided
7. Avoid re-synthesis of hardware solution to accommodate new program binaries.	Avoided

The hardware concept avoids any added latency to the program's execution time. The control flow integrity violation detection analyzes each executed control flow instruction of interest in real-time. A violation is reported, if found, at the end of the execution cycle for the current instruction. For example, if a RET instruction targets an address not on top of the SCS, a violation is flagged on the last cycle of the RET instruction before the next instruction pointed to by the violated PC has a chance to execute.

Integrated into the instruction pipeline, the CFI violation detection system does not require altering the AVR instruction set or modifying existing program binaries to detect control flow violations. The offline metadata collection and security memory usage accommodate new program binaries without re-synthesizing the hardware solution, a trade-off of increased area.

The implemented AVR soft-core, obtained from https://opencores.org, is instruction and timing compatible with the ATMega103 and modified to include four additional parallel ports and watchdog timer. Table III shows the area utilization of the detection system compared to just the AVR soft-core in terms of Look Up Tables (LUTs), Flip-Flops, and Block Random Access Memory (BRAM).

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

TABLE III. CFI VIOLATION DETECTION AREA UTILIZATION

Increase Above AVR Soft-Core			BB CAM
LUTs	***Flip-Flops***	***BRAMs***	***Cells***
98%	171%	8%	1
361%	564%	8%	256

Indicated in Table III, the size of the BB CAM negatively affects area utilization and depends on the number of basic blocks. However, the BB CAM positively supports detection response avoiding latency for the detection of violated ICALL and IJMP instructions. The detection rules search through basic block information in parallel to determine if a target address is a valid function or basic block. Searching each cell in parallel allows detection to complete within the amount of cycles it takes the instruction in question to execute at the cost of increased area.

The detection system underwent a security evaluation based on the Basic Exploitation Test (BET) described in Carlini et al. [6]. The hardware based solution successfully detected hijacked RET, ICALL, and IJMP instructions in the presence of bare metal programs. As expected, hijacked IJMP instructions that targeted an address at the beginning of a basic block went undetected, but instead broke CALL-RET pairing and triggered a RET instruction violation. Additional BET testing in the presence of an RTOS will be complete by the time of the conference.

Additionally, the detection solution did not report false positive detections in the presence of interrupts, setjmp/longjmp function calls on bare metal programs, and basic usage of an RTOS called Femto OS.

V. SUMMARY

The two-tiered concept is categorized by code injection attacks and CRAs. Within each tier, protection is provided through detection, recovery, and tamper evidence. Code injection detection on a Harvard architecture requires the detection of CRAs with an additional instruction integrity checker. Checkpoint rollbacks offer an alternative means to recovery for a system attacked through code reuse if a reset is not desirable. To recover from code injection, a backup copy of the program binary is required. A second copy of the AVR soft-core is used as a sacrificial core to provide a means for on-line data collection of executed malicious instructions.

The presented hardware based solution shows promise to detect most CRAs on low-end embedded systems without added cycle latency at the cost of increased area. Future work includes remaining implementation and testing of proposed system to analyze protection provided versus performance and area tradeoffs.

REFERENCES

[1] Verizon, "2017 data breach investigations report," Verizon, Tech. Rep., 2017.

[2] V. Prevelakis, F. Carmona, C. Oprisa, A. Atzmon, V. Vallero, C. Schlehuber, P. Sifniadis, S. Ioannidis, C. Papachristos, A. Krithinakis, M. Athanatos, F. Rodriguez, M. Heinrich, E. Mar´ın, X. Masip, S. Kahvazadeh, K. Lampropoulos, and A. Bartoli, "Report on taxonomy of the ci environments," CIPSEC Consortium, Tech. Rep., 2018.

[3] Kaspersky Lab ICS CERT, "Threat landscape for industrial automation systems in h2 2017," Online, Kaspersky Lab, Tech Report, Mar. 2018, accessed: May 31, 2018. [Online]. Available: https://ics-cert.kaspersky.com/reports/2018/03/26/threat-landscapefor-industrial-automation-systems-in-h2-2017

[4] CI Cybersecurity, "Framework for improving critical infrastructure cybersecurity, version 1.1," National Institute of Standards and Technology, Tech. Rep., 2018, accessed: Apr. 24, 2018.

[5] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 15–26.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T.R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 161-176, 2015.

[7] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced Control-Flow Integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ACM, 2016, pp. 38-49, 2016.

[8] R. de Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (cfi)," *arXiv preprint arXiv*:1706.07257, 2017.

[9] S. Das, W. Zhang, and Y. Liu, "A fine-grained control flow integrity approach against runtime memory attacks for embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 11, pp. 3193–3207, 2016.

[10] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 847–854, 2010.

[11] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Proc. Automation and Test in Europe Design*, Mar. 2005, pp. 178–183 Vol. 1.

[12] T. Nyman, J. E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 259-284, 2017.

[13] L. Davi, M. Hanreich, D. Paul, A. R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-Assisted Flow Integrity eXtension," in *Proceedings 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2015.

[14] 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.

[15] T. M. Thomas, "Hardware monitors for secure processing in embedded operating systems," Master's thesis, University of Massachusetts Amherst, 2015.

[16] C. Ferguson and Q. Gu, "Self-healing control flow protection in sensor applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 602–616, Jul. 2011.

[17] N. H. Rollins, *Hardware and software fault-tolerance of softcore processors implemented in SRAM-based FPGAs*. Brigham Young University, 2012.

[18] H. M. Pham, S. Pillement, and S. J. Piestrak, "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1179–1192, Jun. 2013.

[19] P. R. C. Villa, R. Travessini, F. L. Vargas, and E. A. Bezerra, "Processor checkpoint recovery for transient faults in critical applications," in *Proc. IEEE 19th Latin-American Test Symp*. (LATS), Mar. 2018, pp. 1–6.

[20] Z. Thomas and S. Abdelwahed, "Active malware countermeasure approach for mission critical systems," in *Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence & Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, 2017 IEEE 15th Intl. IEEE, 2017, pp. 632–638.

978-1-7281-5092-5/19/$31.00 ©2019 IEEE