DEPENDABLE CYBER-PHYSICAL SYSTEMS THROUGH CONTROL FLOW MONITORING

By

Joel Van Der Woude

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Prof. Lui Sha (CS)

May 2013

Abstract

Stuxnet, a virus found in Iranian nuclear facilities, proved the feasibility of an attack on an industrial control system. Similar attacks on critical infrastructure such as the power grid or clean water supply could have enormous economic and political implications. In the interest of preventing future attacks on cyber-physical systems we investigate methods of prevention.

We explore the benefits of making architecture level changes to the microprocessor as well as leveraging the constraints of many embedded systems in order to prevent binary code injection. Our approach utilizes control flow monitoring to ensure correct behavior of the system. By monitoring the control flow we are able to detect an attack on the cyber-physical system and gracefully fail by switching to a trusted Simplex controller.

Our control flow checking mechanism is a stand-alone hardware module implemented with hooks into the pipeline of a LEON3 processor. Our approach avoids incurring high overheads present in software-based approaches, due to the parallel execution of our hardware module. We show that it is possible to detect injected malicious code in such a way that allows us to transition to a trusted Simplex controller in the event of an attack.

Subject Keywords: security; cyber-physical systems; embedded systems; control flow;

Acknowledgments

I would like to thank Professor Lui Sha for his willingness to take me on as an undergraduate researcher as well as his guidance throughout the project. I would also like to generally thank the Dependable Cyber Physical Systems group, this is a collaborative work and it would have been impossible without each and every member of the group.

I would especially like to thank Research Professor Sibin Mohan for introducing me to the work, the time he spent catching me up to speed and providing direction for this project. I am also especially thankful for Fardin Abdi for all the work he has put into the project, both in planning, design and providing guidance.

Stanley Bak and Yi Lu were also important members of this team. Stanley helped out directly with the FPGA implementation as well as providing critical guidance. Yi wrote the critical tool to perform control flow analysis of a binary.

Finally, I would like to thank my friends and family for their love and support throughout my academic career.

Contents

1. Introduction

Stuxnet is a virus that was deployed in nuclear refinement facilities in Iran. The virus used a number of previously unknown vulnerabilities to propagate to various control machines and reprogram the programmable logic controllers used to perform the uranium refinement [18]. Stuxnet has highlighted the possibility and effectiveness of an attack on a nation's critical infrastructure and motivates a closer inspection of the security of industrial control systems everywhere.

Given the tight constraints of many embedded systems, traditional techniques to prevent against cyber attacks are not necessarily feasible. While general-purpose processors are opening doors for new defensive techniques through increased parallelism, clock speed and memory size, many embedded systems are constrained by timing requirements and small on-board memory sizes. New defenses that are designed with the limitations of embedded/real-time systems in mind are required to adequately protect these systems.

Along side the growing importance of cyber security with respect to embedded systems, computer architects are finding themselves unable to keep up with Moore's law, fighting to continue the expected increase in performance. Our research focuses on leveraging this additional chip space to make architecture level changes to microprocessors. With these changes there is opportunity to protect against common cyber attacks that might be leveraged against industrial control systems.

By implementing on-chip control flow analysis; we have found we can detect common attacks such as buffer overflows while introducing negligible overhead on the execution of tasks. We have implemented our proposed control flow graph module on an FPGA in the same fabric as a LEON3 soft-core processor. Using minimal hooks into the pipeline we are able to monitor the control flow of the program. By validating the control flow against the control flow graph generated from the initial binary, we are able to detect attacks which modify control flow.

After detecting attacks we can leverage the Secure Simplex architecture to safely fail, ensuring liveness to the industrial control system. By switching to a trusted controller, we will be able to restore the untrusted controller to a known state and eventually return to the untrusted, high performance, controller.

Section 2 of this paper focuses on the prior research that has been done to make this work possible. Section 3 focuses on the work we have done in this area. Finally, section 4 describes our plans for further work in this area.

2. Literature Review

2.1 Attacks

A survey of embedded processor security enumerates some of the most common vulnerabilities found in embedded control systems. The survey divides the attacks into two types: hardware attacks and software attacks. Hardware attacks are attacks that require physical access to a device while software attacks may be performed remotely [1]. Our research hinges on software attacks and we leave hardware attacks to be mitigated by other means. The major types of software attacks found in embedded systems are: buffer overflows, return-into-libc and code injection attacks [1].

2.1.1 Buffer Overflow

The CWE/SANS list of the top 25 most dangerous programming errors lists buffer overflows as the third most dangerous vulnerability [2]. A buffer overflow is an attack that exploits blocks of memory whose bounds are not checked at runtime. When the bounds are not checked an attacker may write more data into a buffer than was intended [3]. The two most prevalent types of buffer overflows are stack overflows and heap overflows.

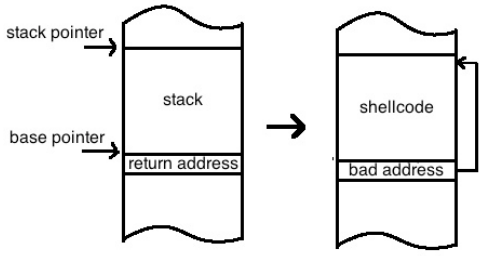

Figure 1 - Structure of the stack before and after a buffer overflow attack.

```
//Will execute unless there is a heap overflow
void good()
{
        printf("You were good!\n");
}

//Will not execute unless there is a heap overflow
void bad()
{
        printf("You were bad!\n");
}

//Set argv[1] to BUFSIZE+4 to overwrite funcptr
int main(int argc, char **argv)
{
        static char buffer[BUFSIZE];
        static void (*funcptr)(void);

        funcptr = (void (*)(void)) good;

        memset(buffer, 0, sizeof(buffer));
        strcpy(buffer, argv[1]);

        (void)(*funcptr)();
}
```

Figure 2 – A program which is vulnerable to a heap overflow by overwriting the value of funcptr.

In a stack overflow, the attacker is able to overwrite the return address of the current stack frame. Often, the attacker will direct execution back into the buffer where he has placed shellcode, machine code written onto the stack to perform some action [3]. Ultimately this allows the attacker to achieve arbitrary code execution. Figure 1 shows the state of the stack before and after an attack.

The other common case of a buffer overflow is called a heap overflow. This occurs when more data is written into a structure on the heap than was intended, modifying the adjacent memory locations. A common use of this vulnerability is to overwrite a buffer allocated on the heap to change the value of a function pointer or other critical variable [4]. Doing so allows the attacker to direct execution to some malicious code. Figure 2 displays the source code of a program with a heap overflow.

It should be noted that in both a stack overflow and heap overflow, an attacker may choose to not redirect execution by overwriting a function pointer or the return address. Instead, the attacker may choose to simply change a value stored on the stack/heap at a higher memory address. By changing a critical value, the attacker may be able to cause some malicious action.

2.1.2 return-into-libc

Return-into-libc is an attack in which an attacker leverages a buffer overflow in order to direct execution towards a function that has been included in the compiled binary. In this way, it does not require the shellcode commonly used in other types of buffer overflows. Instead it can point to a function such as `system()` with an argument like `/bin/sh` [5]. Return-into-libc is a powerful attack because it evades one of the most common defenses against most buffer overflows: a non-executable stack [6].

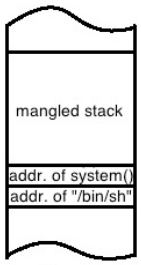

Figure 3 – The state of a stack after an attacker uses a buffer overflow to execute `system("/bin/sh")`.

2.1.3 Code injection

Code injection, in the most general sense, is an attack where the attacker finds a way to trick the target system into executing malicious code. Some of the most prevalent code injection attacks today are SQL injections, however, since embedded systems often don't include SQL databases we focus our attention instead on binary code injection. Binary code injection is most often achieved through a buffer overflow including some binary code written onto the stack, or shellcode. By writing the address of the shellcode into the return address of the stack frame, the attacker is able to execute his shellcode and subvert control of the system. Embedded devices, which are designed using the Harvard architecture, were long thought to be immune to code injection attacks due to their separate instruction and data memories. This belief has been shattered recently by successful code injection attacks on an AVR sensor network [7].

2.1.4 Physical attacks

The survey of embedded processor security lists the following attacks as hardware attacks: evil maid, cold boot, FireWire DMA and bus attacks. The evil maid and cold boot attack are both methods of breaking the encryption and thus confidentiality of a device. The FireWire and bus attacks both allow for an attacker to write data *directly* to memory and execute it, allowing for arbitrary code execution [1].

2.2 Defenses

Historically, PaX and ASLR have been the major defenses against attacks. PaX was the first attempt, implementing an execution “privilege” on memory addresses. This allows the operating system to prevent execution of code on the stack. This, however, can be evaded by heap overflows as well as return-to-libc attacks. The next major technique to emerge was address space layout randomization (ALSR). ALSR randomizes the virtual memory layout of a program in an attempt to prevent attackers from successfully redirecting execution to the right place through a buffer overflow. Techniques to bypass ALSR have been discovered, exploiting the low granularity of randomness to brute force the desired address [14].

There are number of other defenses against buffer overflow attacks [8, 9, 10, 11, 12]. However, we noticed that most of the techniques either did not prevent heap overflows [8, 9, 11, 12], return-to-libc [9, 11, 12] or required a trusted operating system [8, 10, 12]. For embedded systems not running a real-time operating system, another solution would be required.

2.3 Control Flow Monitoring

There has been prior research into verifying the integrity of execution by validating a program's control flow graph (CFG). The aim of control flow integrity (CFI) is to assure that the semantics enforced at the lowest level of execution on the hardware corresponds with the expected behavior as designed by the programmer at a higher level [15].

By considering the series of branches, jumps and calls that traditionally make up a program, it is possible to create a set of states. For each state, there will be a finite set of valid states both preceding and proceeding execution of the state. Restricting the transition of states only to the entry points of states included in the current states set of valid proceeding states allows validation of a CFG. Any transition to a state not included in the CFG is considered to be an attack [15].

Researchers at Microsoft have found a method to guarantee CFI by rewriting a compiled binary to include checks around each control flow operation. Their implementation writes a unique identifier to the beginning of each block of instructions and rewrites the instructions around each control flow operation to verify that the target of the transition has a valid identifier. If the target of a control flow operation does not have a matching identifier, they throw an error and halt execution. They have successfully implemented their model on an x86 architecture running Windows [13].

Microsoft's implementation is especially relevant because of the care they've taken to ensure the guarantees that they provide. Their formal analysis shows that by considering change of states as either a modification to memory caused by an attacker or by the increment of a program counter valid to an address verified by the control flow graph, they can guarantee that the program counter follows the CFG. The following theorem is taken directly from their work [13]:

> Let S_0 be a state with code memory M_c such that $I(M_c)$ and pc = 0, and let $S_1, \ldots, S_n$ be states such that $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$. Then, for all $i \in 0 \ldots (n-1)$, either $S_i \rightarrow_a S_i+1$ or the pc at S_i+1 is one of the allowed successors for the pc at S_i according to the given CFG.

Theorem 1 – Consider M_c to represent the instructions of a program. $I(M_c)$ represents a program which starts up safely such that memory has not yet been modified by an attacker. $\rightarrow_a$ represents a change in memory caused by an attacker. All credit goes to [13, 15].

2.4 Secure Simplex

Simplex is a methodology pioneered by Professor Lui Sha and can be described as "using simplicity to control complexity." The Simplex architecture uses two separate systems to perform a control task. The first system is a high performance but untrusted implementation, allowing for features that are desirable for the task but may have defects. The second system is a high reliability implementation that might not be the highest performance design but is reliable and often verified [16].

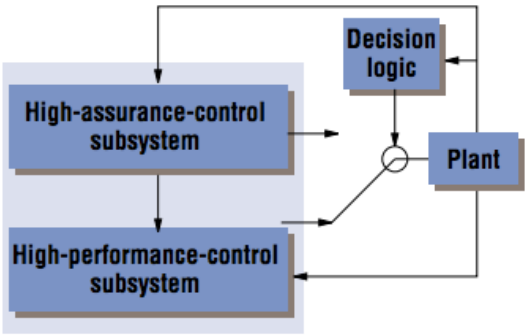

Figure 4 – A diagram of the Simplex architecture. The circle represents the switch that is used to change from the high-performance controller to the high–assurance controller.

Further work on Simplex created what is now called System Level Simplex, which moves the two safety-critical Simplex modules off the control microprocessor to protect against interference from software related faults. This improves the reliability of the Simplex architecture itself in the presence of possibly faulty design of the high performance control system [17].

3. Description of Research Results

3.1 Attack Model

As we considered the attacks listed above, we quickly found it necessary to begin restricting our attack model. First, we decided we would not consider attacks which required physical access. Since our focus is embedded devices, more specifically industrial control systems, physical access raise a whole gambit of options to maliciously impact the system. These options include physical destruction and other methods not particularly interesting to a computer engineer.

Another restriction we have placed is that the system must start safely. What we mean by this is that we do not consider supply chain type attacks where the device was compromised before being able to initialize correctly. Supply chain security is a currently a huge area of research and we defer to the expertise of others on this area. We also assume all key exchanges for any cryptography are successful with no loss of confidentiality.

We do assume that the attacker has networked access to the system. One example of this type of access is an attacker who somehow managed to compromise an employee's machine on the network and now can communicate with the embedded device. Another possibility is an attacker who managed to plant some USB or other device that is networked with the rest of the system.

We also assume that the industrial control system has unknown vulnerabilities which could be leveraged for an attacker. We believe this to be a realistic assumption given the nature of many embedded devices, as they can be running legacy code for decades without being updated or rewritten.

Finally, we assume that our operating system is vulnerable. Again, by nature many embedded systems run legacy code, including operating systems and are not necessarily updated frequently. Due to the continual discovery of vulnerabilities in operating systems, we do not trust the operating system.

3.2 Control Flow Monitoring

After we described our model and evaluated some of the most common attacks on embedded systems, we noticed a common theme, arbitrary code execution. By monitoring the execution flow of a program, and cross validating it against the expected CFG we are able to detect attacks that redirect control flow. We refer back to [13] for the formal analysis of this method.

In order to ensure CFI, it is first necessary to generate a control flow graph. Our approach uses a tool to generate the CFG from the compiled binary. This approach allows for integration with legacy code bases without a rewrite of the code or even possession of the source code. It is also flexible across different architectures with only minor changes required in order to generate the CFG from the binary.

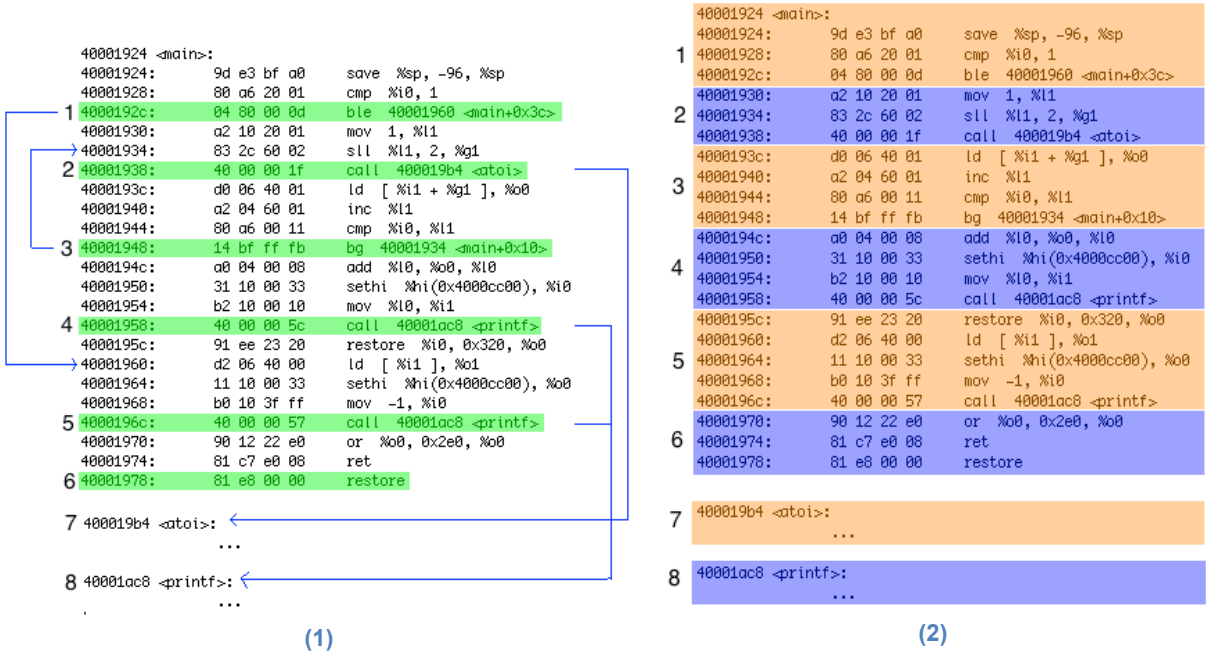

Figure 5 – Figure 5.1 highlights the relevant control flow instructions and their possible destinations. Figure 5.2 highlights the different states of control flow we will store information about.

To generate the CFG, we enumerate all of the control flow instructions in the program: namely variations of branches, jumps and calls. Each control flow instruction is recorded with its relative offset into the code, the number of instructions until the next control flow operation and the index of the proceeding valid control flow instructions along with flag bits to describe whether it is a call or a return. We consider this set of information to describe a state in the control flow.

ID	Address	Instructions	Taken	Not Taken	Flags
1	0x40001924	3	5	2	NULL
2	0x40001930	3	7	7	CALL
3	0x4000193c	4	2	4	NULL
4	0x4000194c	4	8	8	CALL
5	0x4000195c	5	8	8	CALL
6	0x40001970	3	0	0	RET
7	0x400019b4	X	0	0	RET
8	0x40001ac8	X	0	0	RET

Figure 6 – Shows the control flow information taken from the binary for the main function in Figure 5. The flag bits determine if any operation must be done on the hardware stack.

For branches we explicitly list the index of the state if the branch is taken and the index of the state if the branch is not taken. For direct calls and jumps we know that it should always transition to the given state so we set both taken and not taken to the target state. When we do a call or return, our module pushes or pops the return address to the hardware stack, borrowing from SmashGuard, in order to validate on the return [11]. An example binary and it's corresponding CFG information can be found in Figure 5 and Figure 6 respectively.

However, for indirect calls or jumps, that are the result of a function pointer or a jump table, we must enumerate all the possible targets of the instruction. This enumeration requires more in depth analysis and is a point of continued research. A naïve implementation would allow an indirect call or jump to target any memory address. However, this could result in an attacker targeting dangerous targets such as the `system()` call in Unix systems. A tighter implementation would restrict the possible targets to a subset of the states described by the CFG. This will require closer analysis of the binary to determine the valid intended targets.

3.3 Implementation

We have integrated a control flow-checking module into the design of the LEON3 soft-core processor. The LEON3 is a 32-bit synthesizable VHDL model of a SPARC processor. We utilize excess fabric on the FPGA to implement our module with hooks into the pipeline of the LEON3. Our aim was to introduce minimal changes to the design of the LEON3 and we found we could detect changes from the CFG by only hooking into the program counter.

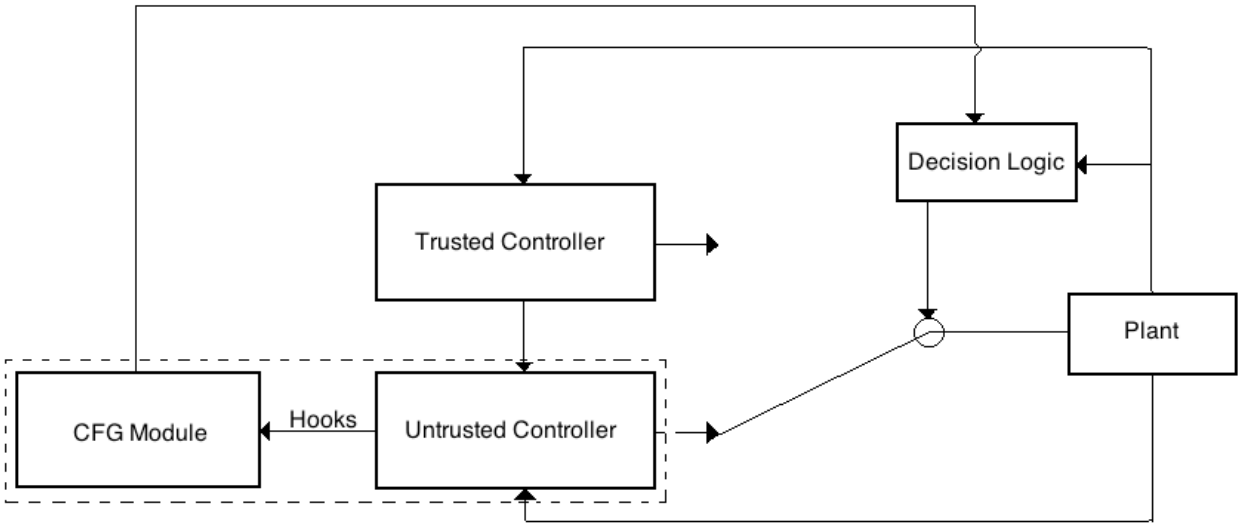

Figure 7 – The overview of how our control flow checking module integrates with the Simplex architecture. The decision block drives the switch (in this diagram represented by the circle) to switch control from the untrusted to trusted controller.

Our module includes a block of SRAM where the description of the CFG is loaded prior to run time. When execution begins, the first block of CFG information is fetched, which enumerates the next valid states. While this state is executing we count the number of instructions executed and pre-fetch the blocks of information for the possible next states. Once we detect that execution has left this control block we check the current address of the program counter against the possible destination addresses. If there is a match we continue execution, otherwise we raise a flag to transition to the trusted Simplex controller.

Below is the basic algorithm used to check the control flow. `pc` is the value of the program counter. `B` is the information that describes a block of control flow, which contains `{pc, n, S}`. `B.pc` is the address at which the control flow block `B` begins. `B.n` is the number of instructions contained in block `B`. Finally, `B.S` is the set of all control flow blocks that may occur after `B`.

```
if pc_current = pc_previous + 1 AND pc_current < B.pc + B.n
      continue;
else if pc_current == B.pc + B.n
      if pc_current ∈ B.S.pc
            B = B.S[i] such that B.S[i].pc = pc_current
            continue;
else
      transition_to_high_assurance_controller();
```

Using the above algorithm we are able to detect unexpected variations in control flow. However, this initial implementation does not correctly respond to function calls. When a function is called, the address at which to return is pushed onto the stack. When the function calls return, it resumes execution at that address. In order to correctly handle call/return instructions we design a stack in hardware using a FILO buffer. This implementation is very similar to SmashGuard [11]. We also modify the control flow blocks B to include members c, r and i flags to mark control flow blocks as having a call or return function as well as the index of the next block.

```
if pc_current = pc_previous + 1 AND pc_current < B.pc + B.n
      continue;
else if pc_current == B.pc + B.n
      if B.c == true
            push B.i
            B = B.S[i] such that B.S[i].pc = pc_current
            continue;
      else if B.r == true
            pop B
            continue;
      else if pc_current ∈ B.S.pc
            B = B.S[i] such that B.S[i].pc = pc_current
            continue;
else
      transition_to_high_assurance_controller();
```

3.4 Analysis

In order to compare the effectiveness of our method with some other common methodologies and related works, we present Figure 6. It is immediately clear that one area of shared weakness is detecting mangled data after a buffer overflow. This is a non-trivial area of work because of the structure of the C language; pointers are not restricted to memory addresses by default. Preventing this type of attack requires validation of each pointer access, ensuring that it falls within the bounds of the high-level object that it represents. There is currently a group of research into this area of verifying safe pointer usage by Professor Iyer and Professor Kalbarczyk at the University of Illinois at Urbana-Champaign.

Technique	Code Injection		Return-to-libc		Overwriting Variables	
	Stack	Heap	Stack	Heap	Stack	Heap
PaX	Y	N	N	N	N	N
PaX ASLR	Y*	Y*	Y*	Y*	N	N
StackGuard	Y	N	Y	N	N	N
SmashGuard	Y	N	Y	N	N	N
ISA Randomization	Y	Y	N	N	N	N
Hardware Support for Control Integrity	Y	Y	N	N	N	N
Control Flow Monitoring	Y	Y	Y	Y	N	N

Figure 8 – List of common attacks enabled through a vulnerable buffer and a number of mitigations against each. An asterisk represents a technique that increases the difficulty of an attack, but does not necessarily prevent an attack from being successful.

Aside from the common weakness against attacks that only mangle specific local variables on the stack, our control flow monitoring technique performs very well when compared against previous techniques. By monitoring control flow we are able to detect all changes from expected behavior. This behavior is expected from the results of [13]. While their approach exists in software our implementation is in hardware. By making changes to the architecture we predict that our technique will have a negligible overhead. This is an important distinction; by avoiding adding additional instructions to the binary we avoid any cost to performance, making it very attractive for use in real-time applications.

4. Conclusion

In conclusion, we have implemented a module to detect changes in control flow that may correlate with attacks. In tandem with the Secure Simplex architecture, this module could be used to detect attacks and fail safely in safety critical applications. Possible uses for these components include critical infrastructure, critical systems within cars and planes, even medical devices.

4.1 Future Work

However, there is much more work to be done in this area. Continued research towards how to extend these techniques from running on a barebones processor to microprocessors running real-time operating systems is needed. We hypothesize that by using additional hooks into the processor to determine current privilege level stored in the MMU, our module would be able to distinguish between multiple tasks and monitor the control flow of each.

Additional research is also needed to determine how the control flow graph size scales with an increasing number of tasks and in the presence of large bodies of code such as a full operating system. Techniques for dynamically generating a control flow graph and loading our module could also create intriguing possibilities for applying our technique to general-purpose processors. Finally, continued research into methods of preventing local variables from being overwritten by a buffer overflow is needed to provide reasonable assurance against that form of attack.

References

[1] A.K. Kanuparthi; R. Karri; G. Ormazabal; S.K. Addepalli, "A Survey of Microarchitecture Support for Embedded Processor Security," *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on* , vol., no., pp.368,373, 19-21 Aug. 2012.

[2] "2011 CWE/SANS Top 25 Most Dangerous Software Errors," Internet: http://cwe.mitre.org/top25/. September 13, 2011. [April 13, 2013].

[3] A. One. (1996, Nov.). "Smashing The Stack For Fun And Profit," *Phrack*, [online] 14(49). Available: http://www.phrack.org/issues.html?issue=49&id=14#article. [April 13, 2013].

[4] M. Conover, "w00w00 on Heap Overflows," Internet: http://www.w00w00.org/files/articles/heaptut.txt. [April 13, 2013].

[5] Nergal. (2001, Dec.). "Advanced return-into-lib(c) exploits (PaX case study)," *Phrack*, [online] 4(58). Available: http://www.phrack.org/issues.html?issue=58&id=4#article. [April 13, 2013].

[6] Solar Designer, "Getting around non-executable stack (and fix)," http://seclists.org/bugtraq/1997/Aug/63. [April 13, 2013].

[7] A. Francillon; C. Castelluccia, "Code Injection Attacks on Harvard-Architecture Devices," *Preceedings of the 15th ACM Conference on Computer an Communications Security*, vol., no., pp.15,26, 27-31 Oct. 2008.

[8] C. Cowan; C. Pu; D. Maier; J. Walpole; P. Bakke; S. Beattie; A. Grier; P. Wagle; Q, Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *(SSYM), Proceedings of the 7th conference on USENIX Security Symposium*, vol. 7, no., pp.5,5, 1998.

[9] M. Milenković; A. Milenković; E. Jovanov, "Hardware Support for Code Integrity in Embedded Processors," *(CASES), Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, vol., no., pp.55,65, 2005.

[10] V. Kiriansky; D. Bruening; S. P. Amarasinghe, "Secure execution via program shepherding," *(SSYM) Proceedings of USENIX Security Symposium*, vol., no., pp.191,206, 2002.

[11] H. Ozdoganoglu; T.N. Vijaykumar; C. Brodley; B. Kuperman; A. Jalote, "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," *IEEE Transactions on Computers*, vol. 55, no. 10, pp.1271,1285, October, 2006.

[12] G. Kc; A. Keromytis; V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," *(CCS) Proceedings of the 10th ACM conference on Computer and communications security,* vol., no., pp.272,280, 2003.

[13] M. Abadi; M. Budiu; U. Erlingsson; J. Ligatti, "Control-Flow Integrity: Principles, Implementations and Applications," *(TISSEC) ACM Transactions on Information and System Security*, vol. 13, no. 4, October, 2009.

[14] H. Shacham; M. Page; B. Pfaff; E. Goh; N. Modad, “On the Effectiveness of Address-Space Randomization,” *(CCS) Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[15] M. Abadi; M. Budiu; U. Erlingsson; J. Ligatti, “A Theory of Secure Control Flow,” (*ICFEM) Proceedings of the 7th international conference on Formal Methods and Software Engineering*, 2005.

[16] L. Sha, “Using Simplicity to Control Complexity,” *IEEE Softwar*e, July 2001.

[17] S. Bak; D. Chivukula, O.Adekunle, M. Sun, M. Caccamo, L. Sha, “The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety,” *(RTAS) Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, 2005.

[18] N. Falliere; L. Murchu; E. Chien, “W32.Stuxnet Dossier,” Internet: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf [April 15th, 2013].