Secure & Resilient Cyber Architectures Conference
MITRE, McLean, VA, October 29, 2010

Automated Preemptive Hardware Isolation of High-Risk Computing Applications

Michael D. Wilder
University of Idaho
Computer Science Department
mwilder@vandals.uidaho.edu

Robert E. Rinker, Ph.D.
University of Idaho
Computer Science Department
rinker@cs.uidaho.edu
(208)885-7378

Jim Alves-Foss, Ph.D.
University of Idaho
Center for Secure and Dependable Systems
jimaf@uidaho.edu
(208)885-5196

***Abstract*—In the face of anticipated persistent cyber attacks it may be desirable to preemptively isolate critical computing applications in hardware. We describe an investigation into providing preemptive hardware isolation by automating the transformation of binaries targeted for general-purpose processors (GPPs) into circuitizable finite state machine with datapath (FSMD) descriptions that are impervious to many vulnerabilities associated with GPPs. These generated FSMDs contain no program memory and do not require access to external data memory, so conventional exploits predicated upon memory manipulation are eviscerated. We show how such descriptions can be used to increase the security and resilience of high-risk applications that would otherwise be deployed as software hosted on a GPP, and discuss situations where this may be desirable. We describe a prototype tool that transforms binaries targeted for the Intel 8051 into circuitizable custom FSMDs, and discuss how the prototype transforms common programmatic constructs in ways that often mitigate and sometimes negate the vulnerabilities contained within the original binaries. The Intel 8051 was chosen for this initial investigation because it is often used in modern USB controllers, and is representative of a class of GPPs that are ubiquitous in embedded systems. We show how interrupt-driven binaries are transformed by the prototype, and discuss techniques used to increase the computational density and decrease interrupt servicing latencies of FSMDs derived from interrupt-driven binaries. Our findings indicate that the prototype is capable of automatically generating circuitizable custom machine descriptions that significantly reduce the likelihood of compromise by conventional remote-origin attacks.**

***Index Terms*—computer security, embedded systems, automatic synthesis, compiler, hardware-software codesign, high-level synthesis, binary translation.**

I. Introduction

In the face of anticipated persistent cyber attacks it may be desirable to preemptively isolate critical computing applications in hardware. There currently exists only one way to accomplish this with sufficient hardware isolation: to create a custom computing machine from scratch. The prohibitive expense associated with such an undertaking can prevent all but the most critical applications from realizing the benefit of hardware isolation. In this paper we describe an investigation into providing preemptive hardware isolation by automating the transformation of binaries targeted for general-purpose processors (GPPs) into circuitizable finite state machine with datapath (FSMD) machine descriptions that are impervious to many vulnerabilities associated with GPPs. The rest of this paper is organized as follows: In Section II-A we describe the preliminary model used to accomplish the automated transformation of programs targeted for GPPs into FSMDs. In Section II-B we show how interrupt-driven binaries are handled under this model, and in Section II-C show how interrupt-driven programs targeted for GPPs can be transformed into circuitizable machine descriptions containing concurrent FSMs that arbitrate for resources on a shared datapath. In Section III we describe a prototype tool that implements the model for programs targeted for the Intel 8051. In Section IV we describe related work, and in Section V we discuss our findings and future research endeavors.

II. Targeted Programs As Custom Computing Machines

Every GPP is endowed with a set of functional capabilities that are bound at the time that it is fabricated. Any program targeted for a given GPP exercises a subset of these capabilities. In essence, any program targeted for some GPP can be thought of as the specification of a custom computing machine couched in terms of said GPP.

For example, imagine that we have some GPP called $\mathcal{G}$ and some program $\mathcal{P}$ targeted for it. $\mathcal{P}$ represents the programmatic form of a solution to some problem. $\mathcal{G}$ contains a set of functional capabilities ($F_{\mathcal{G}}$) that were bound at fabrication time. When $\mathcal{G}$ hosts $\mathcal{P}$ it exercises some subset of functionality ($F_{\mathcal{P}}$) that is available in $F_{\mathcal{G}}$. If we were to observe $\mathcal{G}$ as it is hosting $\mathcal{P}$ and take note of what $\mathcal{G}$ does in order to actualize the solution to the problem, we would obtain a clear idea of the nature of a custom computing machine capable of solving the problem. Moreover, the details of this custom computing machine would be expressed in terms of the functional capabilities and operational semantics of $\mathcal{G}$. In some cases it is possible to obtain enough *a priori* knowledge about $\mathcal{P}$ and $\mathcal{G}$ such that an hardware description language (HDL) description of a custom computing machine functionally equivalent to $\mathcal{G}$ hosting $\mathcal{P}$ can be automatically generated.

A. A Preliminary Model

A program targeted for a GPP is a representation of the solution to some problem as applied to said GPP. When

the GPP begins executing instructions contained within the program, it assumes a sequence of steps on the path toward actualizing the solution represented by the program. This effectively sequential behavior simplifies the development of programs because it emphasizes causality. It is also what allows the creation of a custom FSMD that is functionally equivalent to some program hosted on a GPP.

In the following discussion we use the term *configuration* to describe the logical state of a processor at a specific point in time. Suppose that we wish to create a custom device that is functionally equivalent to some program $\mathcal{P}$ when hosted by $\mathcal{G}$. Let us assume that $\mathcal{G}$ does nothing but host $\mathcal{P}$. $\mathcal{G}$ therefore begins executing $\mathcal{P}$ immediately after it has assumed its initial configuration as a result of a power-on (PON) event, and continues to execute $\mathcal{P}$ until $\mathcal{P}$ completes or another power event occurs. We denote the initial configuration assumed by $\mathcal{G}$ before executing $\mathcal{P}$ as C_0, and each subsequent configuration assumed by $\mathcal{G}$ on the path toward completion of $\mathcal{P}$ is denoted as $C_1..C_N$. The program $\mathcal{P}$ is composed of a sequence of instructions I that dictate the order of operations on $\mathcal{G}$. We denote the initial instruction contained in $\mathcal{P}$ as I_0, and assign some designation I_X to each other instruction contained in $\mathcal{P}$.

Suppose that $\mathcal{P}$ consists entirely of a single loop containing the three instructions I_0, I_1, and I_2. Each of these instructions exercises some functionality provided by $\mathcal{G}$ that is necessary to accomplish the goal of $\mathcal{P}$. All we need to know beyond this is that instruction I_0 is the first instruction to be executed by $\mathcal{G}$. For the sake of this example we shall assume that I_2 is an unconditional branch to instruction I_0, and that I_1 is not a branching instruction.

Given our simple program $\mathcal{P}$ as described above, the sequence of configurations assumed by $\mathcal{G}$ when executing $\mathcal{P}$ is depicted as follows:

$$C_0I_0C_1I_1C_2I_2C_3I_0C_4I_1C_5I_2C_6I_0...$$

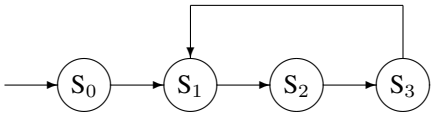
We will hereafter refer to the sequence of configurations assumed by $\mathcal{G}$ when executing the instructions in $\mathcal{P}$ as the *path of $\mathcal{P}$ over $\mathcal{G}$*. Each configuration assumed by $\mathcal{G}$ is assigned a unique number because the effect of any given instruction I on the configuration of $\mathcal{G}$ may not always be the same, even between different executions of the same instruction.

No *a priori* knowledge regarding the configuration assumed by $\mathcal{G}$ after executing a given instruction I is necessary in order to create a custom FSMD that is the functional equivalent of $\mathcal{P}$. It is critical that the configuration assumed by $\mathcal{G}$ after executing instruction I is manifested correctly regardless of variance in the operands. This coherence of configurations is achieved by ensuring that each instruction is implemented correctly in the FSMD that is produced.

We can now provide a partitioning of the path of $\mathcal{P}$ over $\mathcal{G}$ that enables us to create an FSMD that is functionally equivalent to $\mathcal{P}$ over $\mathcal{G}$. Although the path of $\mathcal{P}$ over $\mathcal{G}$ may be arbitrarily long, it contains only three unique instructions. If we regard each unique instruction and the application of the effects of that instruction to the configuration of $\mathcal{G}$ as a state in an FSM, we can define the states for that FSM as follows:

$$[C_0]\ [I_0C]\ [I_1C]\ [I_2C]$$

where the four states in the FSM representing the path of $\mathcal{P}$ over $\mathcal{G}$ are shown in brackets. A depiction of this FSM is shown in Figure 1.



$S_0 = C_0$
$S_1 = I_0C$
$S_2 = I_1C$
$S_3 = I_2C$

Fig. 1. FSM Representing $\mathcal{P}$ Over $\mathcal{G}$

The first state (S_0) in Figure 1 represents the initial configuration assumed by $\mathcal{G}$ after a power event, and therefore has no instruction I that predicates the configuration C_0. Every other state in the graph represents the execution of some instruction I from $\mathcal{P}$, and the application of the effects of that instruction to the configuration of $\mathcal{G}$. In order to complete the FSMD, we must define the datapath that is controlled by the FSM. This is derived by examining each of the instructions in $\mathcal{P}$ and determining which resources of $\mathcal{G}$ are exercised by these instructions. This requires detailed knowledge of the operational semantics of $\mathcal{G}$. It is imperative that knowledge of $\mathcal{G}$ be acquired well beyond that which is available in the instruction set architecture (ISA) in order to ensure that all functionality is implemented correctly in the resultant FSMD.

1) Security Implications: Consideration of the graph in Figure 1 reveals that the implementation of our FSMD does not contain program memory, and will therefore not require logic to fetch or decode instructions. This logic can be omitted from the resulting device because each instruction in $\mathcal{P}$ has been encapsulated in a state within the FSMD. The FSMD always knows its current state and configuration, and can therefore determine its next state. This is important in computer security domains because the set of states that can be assumed by the resultant FSMD is known and fixed, and the logic used to provoke a transition of state is bound into the computing fabric of the resulting device and is therefore immutable.

It is likewise the case that custom FSMDs generated using this technique do not require access to external data memory. Any data memory used by $\mathcal{P}$ can be integrated into the same circuitizable FSMD description representing $\mathcal{P}$ over $\mathcal{G}$. This is important in computer security domains because the original program $\mathcal{P}$ is transformed into a custom, self-contained computing machine that cannot be compromised using conventional attack vectors. The resulting device will also be capable of operating at a higher frequency because no off-chip memory accesses are required.

It is also important to note that knowledge acquired by an adversary regarding exploits targeting the ISA of $\mathcal{G}$ is no longer of any value when attacking an FSMD generated

by this technique. This is because the generated FSMD is customized to the needs of a particular program $\mathcal{P}$, and only those functional capabilities of $\mathcal{G}$ necessary to host $\mathcal{P}$ are considered when generating an FSMD of $\mathcal{P}$ over $\mathcal{G}$. None of the architected resources available in $\mathcal{G}$ are themselves architected in the resultant FSMD, and the operation of the resultant FSMD can be affected only by a device that is physically connected to its interface. This dissolution of the ISA characteristics of $\mathcal{G}$ is due to the fact that this technique regards the binary targeted for $\mathcal{G}$ as an intermediate representation (IR) in a sequence of IRs that ultimately lead to the generation of a custom FSMD description.

2) Branching and Invocations: Absolute and conditional branches are handled in a straightforward way under the model. Absolute branches are a simple transition of state. Conditional branches are handled by a test of that portion of the datapath upon which the given condition is founded as per the operational semantics of the host GPP.

Invocations (subroutine calls) are often implemented as enhanced branching instructions. A simple means of handling invocations requires some form of last-in-first-out (LIFO) mechanism in the resulting FSMD to store the return destination. This replicates the behavior of the typical GPP. Because our goal is to describe a custom computing machine, however, such a simplistic approach can be undesirable.

3) Recursion: Recursion is useful when describing the solution of certain classes of problems. Care must be taken to ensure that the limitations of the recursive portions of the solution are thoroughly understood before executing the solution on a GPP or implementing the solution as a custom FSMD. Creating a custom FSMD that is functionally equivalent to a program containing recursion can be accomplished in multiple ways.

One way to approach the problem is to replicate the invocation resources available on the GPP in the resultant FSMD. In this case the FSMD will exhaust its runtime resources if and only if the host GPP exhausts its runtime resources. A disadvantage to this approach is that the spatial requirements imposed by replicating the invocation resources available on $\mathcal{G}$ can be difficult or impossible to accommodate when creating an FSMD destined for a space-constrained environment.

Another approach is to allow the designer to specify the resources required for recursion in the FSMD. The designer can then use the generated FSMD to ensure that the resource limits are correct.

B. Interrupts

Interrupt-driven software is a vital component of modern digital systems. In this section we describe a model for producing custom FSMDs from interrupt-driven software. Details concerning the implementation of interrupt mechanisms on specific GPPs are ignored so that we can discuss generally how to transform interrupt-driven programs into custom FSMDs.

A program containing interrupt-driven components can be modelled as co-operating FSMs sharing a common datapath. In this model the main thread of execution and each interrupt service routine (ISR) is a separate FSM that performs work when it possesses the execution quantum. Consider the program $\mathcal{P}_1$ in Figure 2. This program contains two ISRs named `isr_inc` and `isr_dec`, and the main thread of execution. The main thread simply initializes registers `r1` and `r2`, enables interrupts, and continually transfers the values of `r1` and `r2` to ports `p0` and `p1` before incrementing the value of `r2`. The purpose of `isr_inc` is to increment `r1` and then perform the logic necessary to effect a return from the interrupt. The purpose of the `isr_dec` ISR is to decrement `r1` and then perform the logic necessary to effect a return from the interrupt.

```
    entry:          ; entry point
1.      ljmp work
    org 3
    isr_inc:        ; irpt 0 ISR
2.      inc r1
3.      reti
    org 19
    isr_dec:        ; irpt 1 ISR
4.      dec r1
5.      reti
    org 256
    work:
6.      mov r1,#0   ; init r1
7.      mov r2,#0   ; init r2
8.      mov ie,#85h ; enable irpts 0,1
    loop:
9.      mov p0,r1   ; emit r1
10.     mov p1,r2   ; emit r2
11.     inc r2
12.     sjmp loop
```

Fig. 2. Program $\mathcal{P}_1$

The three units of the program in Figure 2 are represented as three discrete FSMs shown in Figure 3. State S_0 appearing as the initial state in each ISR is a synthetic state that is added to denote a ready state during which no processing of functionality present in the ISR is performed. This synthetic initial state is analogous to the initial state of the main thread in program $\mathcal{P}_1$. In each case, S_0 is the state assumed by the FSM before any instructions representing functionality contained in the software from which the FSM was derived are executed.

The three FSMs appearing in Figure 3 are separate machines. Although a state named S_0 appears as the initial state of each of these machines, each S_0 is a distinct state from which a single transition to another state in the same machine can be made. In Figure 3, for instance, the FSM representing the ISR `isr_dec` can only transition from state S_0 into state S_4.

In order to correctly model the interrupt-driven program $\mathcal{P}_1$ found in Figure 2, we require a means of ensuring that only

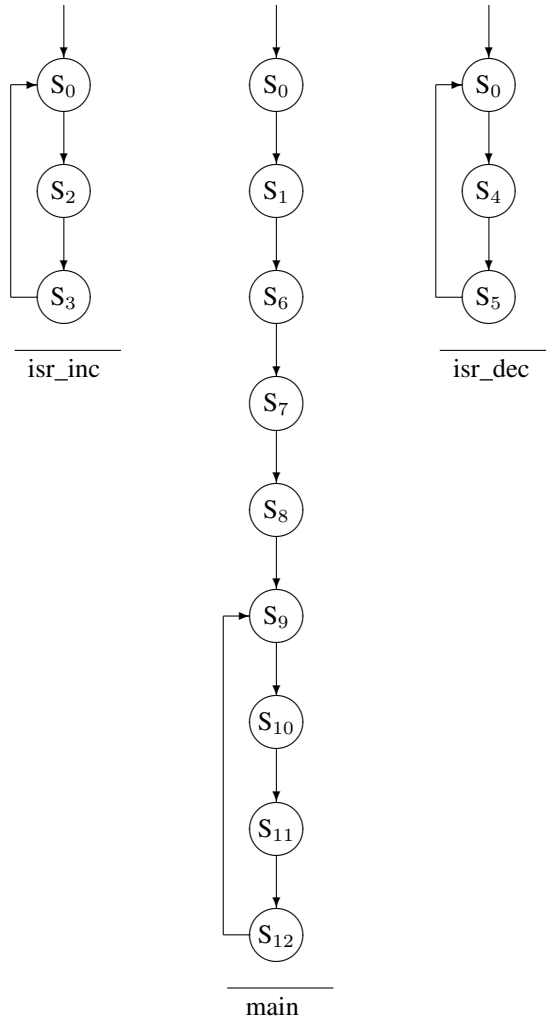


Fig. 3. Units of $\mathcal{P}_1$ Over $\mathcal{G}$ as FSMs

one of the FSMs in Figure 3 is active at any given clock cycle during the operation of the resulting FSMD representing $\mathcal{P}_1$ over $\mathcal{G}$. We also must ensure that the correct FSM is activated at any given clock cycle during the operation of the resulting FSMD. These functions are accomplished by an entity called the executive arbiter (EA).

The role of the EA is to determine which FSM will execute at any given clock cycle. The method employed by the EA to determine which FSM will execute is dependent upon the interrupt mechanism implemented on the target GPP. The EA uses information gathered from interrupt sources, interrupt status registers, and the interrupt mechanism employed by the target GPP to select which FSM will be granted the quantum for a given clock cycle. Once an FSM has been granted the quantum, it uses its current state, coupled with the current configuration of the FSMD, to determine the logic that should be exercised and to potentially transition state.

As an example, assume that we are executing the program depicted in Figure 3. Assume that this device is initially being held in reset. No transitions or levels of external stimuli matter, and no FSM in the device can be granted a clock cycle because the EA is being held in reset. When the reset line to the device becomes deasserted, the device is initialized, which includes setting all FSM state registers to their respective S_0 state. Once initialization has completed, the EA begins the arbitration process. At this point the wiggling of any interrupt sources does not matter because device-specific interrupt enabling appearing in program $\mathcal{P}_1$ at state S_8 has not yet occurred, so the EA will grant the clock cycle to the `main` FSM. The EA will continue granting clocks exclusively to the `main` FSM until it has completed S_8. Once the `main` routine has enabled interrupts in state S_8, combinational logic tied to the interrupt lines that source the interrupts will be used to update interrupt status registers internal to the FSMD. At each clock the EA will use the information contained in the interrupt status registers, along with the interrupt mechanism employed by the host GPP, to determine which FSM will be granted the ability to exercise logic.

The arbitration performed at each clock cycle ensures that only the correct FSM is permitted to perform work at any given time. The particulars of what constitutes correctness in this case are determined by the algorithm employed by the GPP for which the software was targeted.

C. Concurrency

We have devised a method whereby existing software targeted for sequential, synchronous execution on a given GPP can be transformed into a synchronous FSMD containing concurrently executing FSMs sharing a datapath. This method requires no modification of the original program.

Consider program $\mathcal{P}_1$ in Figure 2. This program is composed of three distinct units of execution: the `main` unit, and two units represented by the ISRs named `isr_dec` and `isr_inc`. Examination of this program reveals that each of these units modifies register `r1`. We wish to create an FSMD wherein all of these execution units are operating concurrently. We therefore need to devise a means wherein coherency of the resource `r1` is maintained.

A straightforward way to ensure coherence of `r1` is to guard the resource with an arbiter. Such devices are commonly employed in digital systems, and can be designed into the datapath of an FSMD. The arbitration logic employed by the device must be expressed in such a way as to ensure exclusivity of access while preventing starvation. Once we have devised such an arbiter, we can design our datapath in such a way that all access to the resource under guard (`r1`) is permitted only when granted by the arbiter. We then inject states into the FSM of each device that might access the resource under guard. These synthetic states represent the requesting of access by an FSM, waiting for the grant to be received from the arbiter, and relinquishing of the grant. The FSM representing the `isr_dec` unit of $\mathcal{P}_1$ appearing in Figure 3 would appear as shown in Figure 4 after being transformed in the manner described.

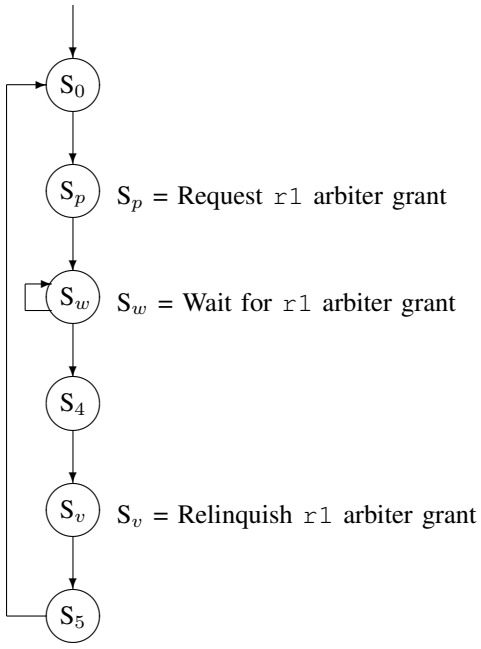


Fig. 4. Unit isr_dec of $\mathcal{P}_1$ Over $\mathcal{G}$ as Concurrent FSM

The synthetic states appearing in the FSM of isr_dec ensure that the FSM does not access `r1` unless it has been granted access from the arbiter guarding `r1`. The proper design of this arbiter ensures exclusivity of access to `r1`. Since all units of program $\mathcal{P}_1$ access `r1`, each of the other FSMs must be modified to observe the same access protocol illustrated in Figure 4.

When all FSMs have been modified in the manner described, there remains no further need for arbitration of execution among the FSMs in the device. The per-cycle arbitration that was imposed upon the co-operating FSMs has been reduced to the per-resource arbitration of concurrently executing FSMs. Each unit depicted in Figure 3 will act independently, exercising logic on every clock cycle. Each FSM that was derived from an ISR in the program will have exclusive access to the interrupt source that was previously monitored by the EA and used to initiate execution of the ISR. To put it another way, each unit representing an ISR will now monitor its own source and engage in the work represented in the ISR on its own volition. Each ISR is also responsible for "resetting" itself and returning to the ready state upon completion of the logic contained in the ISR. The details of this housekeeping are dependent upon the GPP for which the ISR was originally targeted.

We have now transformed what was originally a program containing a main thread and two ISRs into an FSMD containing three concurrently executing units. We were able to easily determine the hazards contained within program $\mathcal{P}_1$, but it is not always possible to easily identify hazards in a program. In the case where the designer has intimate knowledge about the operation of and the resources used by $\mathcal{P}$, she can provide enough information to ultimately generate the FSMD from $\mathcal{P}$. The designer can then use the generated FSMD to verify that the specification of the hazards in $\mathcal{P}$ was correct.

III. The MADGEN Prototype

We have created a prototype tool that transforms programs targeted for the Intel 8051[6], [7], [8] into circuitizable FSMDs. The MADGEN (MAchine Description GENerator) prototype takes as inputs the ROM image of an 8051 program in Intel .hex format and an optional set of constraints. The MADGEN prototype produces a circuitizable VHDL description of an FSMD that is functionally equivalent to the program as hosted on the 8051. The prototype does *not* partition the input program into a component hosted on a GPP and a circuitizable component; MADGEN transforms the entire input program into a single, self-contained, circuitizable machine description in order to achieve a level of hardware isolation sufficient for even the highest risk computing applications.

Because MADGEN analyzes the ROM image of the input program, the user is free to employ any toolchain capable of producing an .hex image for the 8051. MADGEN is designed to make it easy to implement bindings for hosts other than the 8051. The Intel 8051 was chosen for this initial investigation because it is often used in modern USB controllers, and is representative of a class of GPPs that are ubiquitous in embedded systems.

A. Operational Overview

When presented with the ROM image of an 8051 program, MADGEN analyzes it to determine the nature of the program. MADGEN creates a control-flow graph (CFG) representing the main thread of execution of the input program, and then decorates this CFG with dataflow information gleaned from performing abstract interpretation of the main thread. If analysis of the main thread reveals that interrupts are being used, MADGEN will create a separate CFG for each ISR and perform abstract interpretation of the ISR. MADGEN then performs a sequence of analyses and transformations to ultimately produce an FSMD representing the input program. The user is able to guide MADGEN in this process via constraints specifying the analyses and transformations to be performed. MADGEN treats each thread in an interrupt-driven binary as a discrete computing machine that resides on a shared computing fabric. If the user wishes to transform an interrupt-driven 8051 program into an FSMD containing concurrently executing units as described in Section II-C, MADGEN requires that the user provide a set of hazard constraints.

IV. Related Work

Converting software binaries into machine descriptions of varied form is an active research area [1], [2], [3], [4], [5], [9], [10], [11], [12], [13]. However, each of these efforts performs a partitioning of the input binary into a component that will be hosted on a GPP and a component that is circuitizable. This approach is insufficient where computer security is a

governing concern because it does not provide an adequate level of application isolation. Moreover, the prevailing mindset of these efforts is to reduce an input binary into "kernels" of computationally intensive functionality that are circuitized, with the use of these kernels being controlled and coordinated by software hosted on a GPP. This is undesirable from a security perspective because overall control of the resultant product depends upon the integrity of software hosted on a GPP.

V. Conclusions

We have demonstrated that automated preemptive hardware isolation of critical computing applications can be provided by transforming binaries targeted for GPPs into circuitizable FSMDs. Because the FSMDs generated by this process contain no program memory and do not require access to external data memory, the level of isolation provided is such that vulnerabilities predicated upon the manipulation of memory are effectively eliminated. Because the method used to generate these FSMDs is automated, the temporal investment required to build a custom computing machine from a binary targeted for a GPP is significantly reduced. This method will therefore increase the domain of computing applications that can benefit from preemptive hardware isolation.

We have described a technique for transforming interrupt-driven binaries into synchronous FSMDs containing concurrently executing units that arbitrate for resources on a shared datapath. This technique reduces the coarse-grained interleaved execution employed on GPPs to fine-grained arbitration for specific resources without the need for swapping context between executing units. This facilitates the rapid development of applications that leverage the concurrency inherent in the FPGA and will be useful in reducing interrupt servicing latencies in time-critical GPP applications deployed on the FPGA.

We have created a prototype (MADGEN) that transforms programs targeted for the Intel 8051 into circuitizable FSMDs. MADGEN generates the description of custom, self-contained, circuitizable computing machines that are impervious to many vulnerabilities associated with GPPs. Experiments conducted with MADGEN indicate that for some 8051 programs it generates an FSMD that uses less area and is capable of operating at a higher frequency than the soft implementation of an 8051 processor.

References

[1] Zhi Guo, Walid Najjar, and Betul Buyukkurt. Efficient Hardware Code Generation for FPGAs. *ACM Trans. Archit. Code Optim.*, 5(1):1–26, 2008.

[2] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, page 461, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Eeconfigurable Architectures. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 507–512, New York, NY, USA, 2000. ACM.

[4] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 163–174, New York, NY, USA, 2006. ACM.

[5] Gaurav Mittal, David C. Zaretsky, Xiaoyong Tang, and P. Banerjee. Automatic Translation of Software Binaries Onto FPGAs. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pages 389–394, New York, NY, USA, 2004. ACM.

[6] Philips Semiconductor Corporation. *80C51 Family Architecture*, March 1995.

[7] Philips Semiconductor Corporation. *80C51 Family Hardware Description*, September 1997.

[8] Philips Semiconductor Corporation. *80C51 Family Programmer's Guide and Instruction Set*, December 1997.

[9] Greg Stitt, Roman Lysecky, and Frank Vahid. Dynamic Hardware/Software Partitioning: A First Approach. In *DAC '03: Proceedings of the 40th Conference on Design Automation*, pages 250–255, New York, NY, USA, 2003. ACM.

[10] Greg Stitt and Frank Vahid. Hardware/Software Partitioning of Software Binaries. In *ACCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 164–170, New York, NY, USA, 2002. ACM.

[11] Greg Stitt and Frank Vahid. Binary Synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–30, 2007.

[12] Frank Vahid, Greg Stitt, and Roman Lysecky. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. *Computer*, 41(7):40–46, July 2008.

[13] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector Processing as a Soft Processor Accelerator. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):1–34, 2009.