Hardware-Assisted Code Obfuscation for FPGA Soft Microprocessors

Meha Kainth, Lekshmi Krishnan, Chaitra Narayana, Sandesh Gubbi Virupaksha and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA, USA

***Abstract*—Soft microprocessors are vital components of many embedded FPGA systems. As the application domain for FPGAs expands, the security of the software used by soft processors increases in importance. Although software confidentiality approaches (e.g. encryption) are effective, *code obfuscation* is known to be an effective enhancement that further deters code understanding for attackers. The availability of specialization in FPGAs provides a unique opportunity for code obfuscation on a per-application basis with minimal hardware overhead. In this paper we describe a new technique to obfuscate soft microprocessor code which is located outside the FPGA chip in an unprotected area. Our approach provides customizable, data-dependent control flow modification to make it difficult for attackers to easily understand program behavior. The application of the approach to three benchmarks illustrates a control flow cyclomatic complexity increase of about 7× with a modest logic overhead for the soft processor.**

***Keywords*-Soft microprocessor, code obfuscation**

I. Introduction

As FPGAs are deployed in a greater number of embedded platforms, intellectual property protection becomes more important. For example, field-programmable devices are now found in mobile platforms, automobiles, and a variety of consumer products. In most cases, program execution requires access to instructions located in unprotected, off-chip memory providing an inviting target for attackers to observe both program instructions and the order in which they are accessed. Although encryption provides significant protection for soft processor instructions, the use of encryption does not mask control flow and can expose repetitive instruction address patterns to an attacker. These patterns can leak algorithm information even if the specific instructions are not decipherable.

As a supplement to encryption, code *obfuscation* [1] can be used to further inhibit program instruction evaluation. The goal of obfuscation is to retain the same code functionality, but mask code behavior to make it much more difficult for an attacker to discern software algorithm function. Many different types of software obfuscation have been developed, including obfuscated code layout, data structure masking, and control flow obfuscation [2]. Our work focuses on the third type of obfuscation.

Unlike fixed-functionality processors, an FPGA which contains a soft processor has the capability to be configured at the hardware-level on a per-application basis. While processor operation and the processor instruction set are unaffected, slight changes in the control flow of the processor can augment traditional software obfuscation techniques to make obfuscation more confusing to the attacker. *The main contribution of our approach is the use of a small hardware module isolated in the FPGA to determine control flow for code retrieved from external memory.* A small hardware module can be easily created at application compile time and isolated in a reserved region of the FPGA that potentially could be changed via partial reconfiguration. Reconfiguration allows for replacement of this module at FPGA load time in a time period which is small compared to application load time. The contents of the partial bitstream can be secured by bitstream encryption, providing protection for the obfuscation hardware against attackers.

Our approach to code obfuscation takes advantage of control flow flattening [2] [3] to automatically collapse the control flow of C language procedures. The customized obfuscation hardware module inside secure FPGA hardware allows for correct program control flow. Through experimentation with obfuscation we show that the code complexity not only increases substantially, but since all control flow information is not available to the attacker, the ability to understand the obfuscated code is significantly hampered. To verify the functionality of our approach, a complete software system has been created and tested on an Altera DE4 board. Our customized *branching function* is added to C code which has its control flow flattened to obfuscate program behavior from an attacker with access to the external memory bus. For FPGA implementation, the branching function is implemented in hardware structures with very low overhead, out of sight from an attacker. In experiments targeted to a Stratix IV FPGA it was found that our code obfuscator increases code flow cyclomatic complexity by a factor of 7 for a modest lookup table (LUT) count increase.

The remainder of the paper is organized in the following fashion. Section II provides background on code obfuscation and its use in securing software. Section III provides an overview of our obfuscation algorithm and our implementation approach. Section IV describes our experimental approach and results are presented in Section V. Section VI provides conclusions and directions for future work.

II. Background

A. Code Obfuscation

Code obfuscation is typically employed to mask the specific details of program flow and function from an attacker. The algorithmic function of a program is retained, although its structure and efficiency are typically altered. Quantitatively, a goal of code obfuscation is the development of a polynomial time obfuscation algorithm that requires at least an exponential deobfuscation effort by an attacker [1].

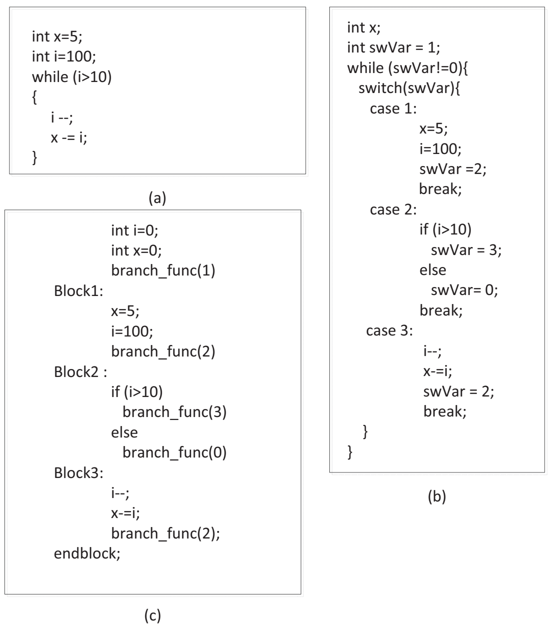
To obscure dynamic (run-time) control flow analysis, it is desirable to create multiple copies of basic blocks which perform the same function but have been coded in different styles. A value is then used in a conditional statement to select specific equivalent paths [1]. Another approach focuses on distributing decision-making *predicate* variables used in conditional statements across multiple functions [4]. Our approach uses predicate variables but hides the function used to control program branching in FPGA logic. Another common code obfuscation technique involves adding junk code or additional program states to obscure program analysis. Often, code is added near function calls or conditional instructions [5] to obscure control flow. The use of indirect branch targets makes analysis particularly difficult. Adding irrelevant conditions with locally generated data to conditional statements leads to an expansion of program state space [2]. This approach is particularly effective if the consumed data is not easily observed by the attacker. Our implementation uses this approach in addition to hardware-assisted branching functions to diversify control flow options.

Other code obfuscation techniques include self-modifying code [6], mobile code download [7], and software diversification [8]. The latter two approaches focus on the periodic use of different versions of code for network-connected applications. These techniques are beyond the scope of this work.

B. Control Flow Flattening and Branching Functions

An effective way of obscuring code execution is to *flatten* a procedure into a series of basic blocks which reside within a loop and switch statement combination (Fig. 1) [2] [3]. As a result, the previous and next basic blocks of computation are much less apparent to an attacker. This type of structure has several critical aspects. First, the next executed basic block in the loop is controlled by a computation in the current block. In this simple example, the switch control variable $swVar$ is set to a fixed constant value in the current basic block although, as discussed in the next section, much more complicated assignments can be used. Second, the termination of the computation is made much less apparent to an attacker. Since basic blocks can be positioned in any order in the high level and assembly code, the sequence of execution can be effectively obscured.

Although it has been shown that analyzing flattened programs is an NP-hard problem [2], in the years since the initial introduction of this obfuscation idea, a number of enhancements have been proposed to improve the level of



```
int x=5;
int i=100;
while (i>10)
{
   i --;
   x -= i;
}
```

(a)

```
int x;
int swVar = 1;
while (swVar!=0){
  switch(swVar){
    case 1:
            x=5;
            i=100;
            swVar =2;
            break;
    case 2:
            if (i>10)
              swVar = 3;
            else
              swVar= 0;
            break;
    case 3:
            i--;
            x-=i;
            swVar = 2;
            break;
  }
}
```

(b)

```
          int i=0;
          int x=0;
          branch_func(1)
     Block1:
          x=5;
          i=100;
          branch_func(2)
     Block2 :
          if (i>10)
            branch_func(3)
          else
            branch_func(0)
     Block3:
          i--;
          x-=i;
          branch_func(2);
     endblock;
```

(c)

Fig. 1. Example of control flattening showing original code (a), flattened code (b), and obfuscated code (c)

obfuscation provided by control flow flattening. Perhaps the most effective enhancement [9] to control flow flattening has involved the replacement of the static switch variable assignment (e.g. $swVar = 2$) with a *branching function* of the form:

$$branch\ target = f(input\ data, path\ signature) \quad (1)$$

In the equation, the path signature represents a combination of the start addresses of the basic blocks that have been executed in sequence by the procedure. Data values can be used to guide branching towards multiple equivalent copies of the same target basic blocks.

Our approach to code obfuscation enhances these previous techniques by isolating the branching function *in hardware* on a per-application basis with almost no changes to the soft microprocessor pipeline. This small hardware function can be updated on a per application basis (potentially using partial FPGA reconfiguration) to completely hide portions of the code execution control flow from the attacker. Zhuang, *et al.* [10] previously suggested using hardware-assisted obfuscation to dynamically reposition chunks of virtual memory during code execution. The need for an operating system may not be appropriate for embedded applications implemented on FPGA-based soft processors.

C. Security Model

It is assumed that the FPGA hardware and its associated bitstream are secured using bitstream encryption, which makes it difficult for all but the most experienced attackers to decipher a bitstream using hardware-oriented means such as side-channel attacks. External memory, which holds soft processor instructions and its bus interface to the FPGA, are insecure. Data memory values used by the processor can either be located in internal FPGA memory or in external DRAM. It has been previously shown [10] that neither encryption nor caches remove the benefits of code obfuscation. Although encryption hides the values of the instructions, instruction address patterns are still apparent. Obfuscation scrambles these patterns making it difficult to determine program function. Attacks which force cache misses so that all instructions are fetched from external memory are also possible. Code obfuscation is *complementary* to code and data encryption and provides protection for control flow which encryption does not provide.

III. Hardware-based Obfuscation

Our code obfuscation approach operates in a series of automated steps (Fig. 2) that are customized for FPGA implementation of obfuscation hardware.

Control flow flattening - The approach first involves flattening the control flow of each selected C procedure into a single *while* loop and *switch* statement. Control flattening is performed on each selected C procedure, as illustrated in Fig. 1(a) and (b). The flattening of code into a format shown in Fig. 1(b) takes place in a series of steps. First, all *do* and *for* loops are converted to *while* loop equivalent representations. Subsequently, all *if*, *while*, and *switch* statements are converted to the *while/switch* structure illustrated in Fig. 1(b). The algorithm implementation uses a stack to keep track of nested loops and conditionals.

Branching function generation - A hardware implementation of a branching function is generated which dynamically determines the next basic block to be executed by the procedure. This action not only requires the generation of a small amount of hardware, but also the modification of the flattened C code to include subroutine "calls" to the *branching function* which pass control to the hardware. The specific hardware branching function is customized on a per application basis based on user specification. In our current implementation, the branching function uses information regarding the execution path of the program (e.g. former branching function calls) and specific data values from the datapath to generate a next target address.

Backend compilation - The binary for the obfuscated code is generated by a software compiler while the branching function is synthesized for FPGA implementation.

We now explain each step of our flow in greater detail. Specific steps in both the hardware and software compile flows are shown in Algorithms 1 and 2.

A. Code Modification for the Branching Function

The flattened C code is modified to insert calls to a branching function (Fig. 1(c)) which determines the execution of the next basic block. During compilation, the subroutine call instructions are converted to MIPS jump-and-link (jal) instructions which include the address *branch_func*, a dummy procedure in the code which performs no useful function. The jal assembly instruction related to this call is identified in the microprocessor as it executes and a hardware branching function (Fig. 3) is activated to determine the branch target. As shown in Fig. 1(c), all call statements include a data value. Although the example shows constant values as a call argument, input values could be used instead to allow for varying branch target assignment. The branching function uses the **address** of the jal instruction in determining the target address. Since the calls are in different locations in the code, their resulting targets will be different.

Algorithm 1 Detailed Automated Software Compile Steps

```
1: C code is flatten (Fig. 1(b)).
2: Flattened C code compiled using MIPS C compiler.
3: Static analysis of code using sample data sets to determine
   branch points and targets.
4: Flattened C code is modified to include calls to branching
   function, $branch\_func$. A "dummy" code stub is inserted
   in the code for $branch\_func$.
5: Modified code is compiled using MIPS C compiler
6: For each code path, the sequence of addresses where calls
   are made to $branch\_func$ is determined using the static
   analysis information.
```

Algorithm 2 Detailed Automated Hardware Compile Steps

```
1: The user specifies $branch\_func$ in Verilog.
2: An index for each branch in each code path is generated
   using the results from step 6 of Algorithm 1 and (2).
3: For each branch target in a specific code path, the branch
   offset is found using the index determined in the previous
   step.
4: RTL for the lookup table which associates branch offsets
   with indices is generated.
5: Obfuscation hardware, including the branching function
   and the lookup table, is synthesized and merged into the
   remaining bitstream for the processor.
```

B. Building the Branching Function in FPGA Hardware

To defend against dynamic replay attacks, not only is the branching function implemented in FPGA hardware, but additional hardware is included which determines the next basic block branch target for the code (effectively, the next case in the switch statement). Additionally, unlike the code shown in Fig. 1(b), the next target basic block is determined by the branching function using a combination of input data and previous control flow (path signature) information. The use of data allows for different control flows for the same function based on data values This approach makes replay attacks using varying data values more difficult to understand.

The structure of our obfuscation hardware is shown in Fig. 3 as shaded blocks. A trigger for the obfuscation hardware is provided when a *call* instruction to the branching function in software (e.g. a *jal* instruction to a specific branch address) is fetched from the instruction memory. An instruction match is identified with a comparator (e.g. the =? block) which

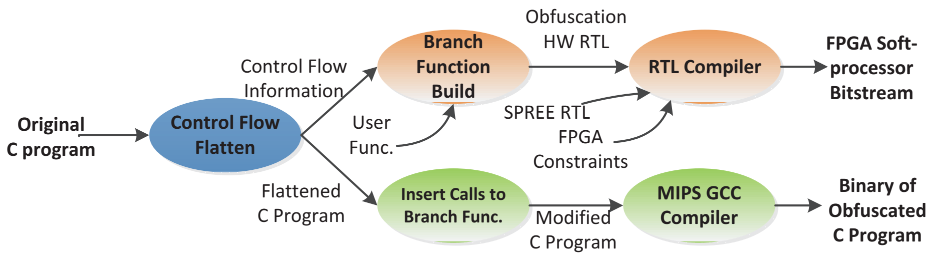


Fig. 2. Flow diagram showing the steps necessary to implement hardware-based code obfuscation

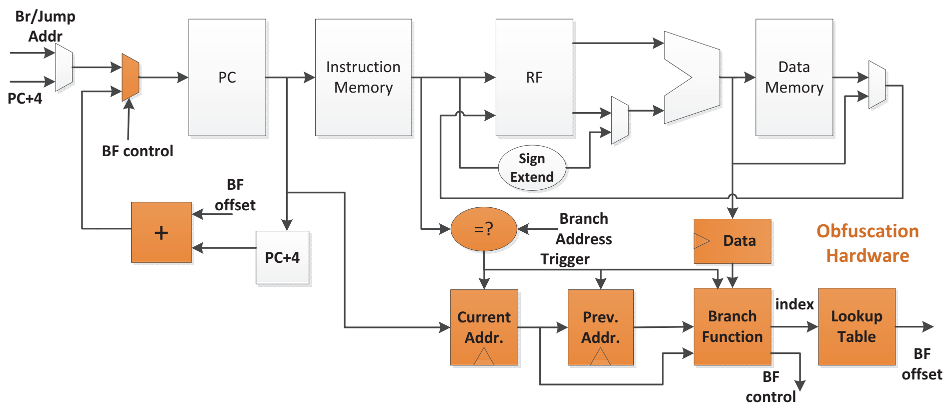


Fig. 3. MIPS datapath augmented with hardware to perform code obfuscation. Note that some details of the datapath have been omitted.

provides a trigger to start the control transfer to the obfuscation hardware. Although virtually any combination of processor information could be used to determine branch targets via the branching function, we use three inputs for testing: the *address* of the current *call* instruction (**current address**), the address of the previous *call* instruction (**previous address** - which was formerly the current address before the current *call*), and data values. In our initial implementation, the branching function generates a 32-bit output index using the 9 low-order bits of the previous address, the 14 low-order bits of the current address, the 7 high-order bits of the current address, and 2 data bits. In general, any ordering or function of these input bits could be used to generate the index. Thus, our branching function can be specified as follows:

$$index = \{paddr[8:0], caddr[13:0], caddr[31:25], data[1:0]\} \tag{2}$$

where $caddr$ is the **current address** and $paddr$ is the **previous address**. Virtually any function is possible, included more complicated ones.

Following the generation of an index, a lookup is made in a lookup table which operates as a content-addressable memory. Effectively, each valid entry in the lookup table generates a *branch offset* for the current program counter (PC). As seen on the right side of Fig. 3, the offset is added to PC+4 to generate a new target address. A control signal from the branching function indicates the selection of the new target address. The creation of the obfuscation hardware (as shown in Fig. 2) involves the population of the branching function and lookup table blocks. The final two steps in the automated obfuscation process involve converting the modified code (e.g. Fig. 1(c)) into binary code and the synthesis of the soft processor and obfuscation hardware using a standard MIPS GCC compiler and FPGA RTL compiler, respectively. If only the obfuscation hardware is modified, it can be synthesized separately into a constrained region and bitstream-merged into the soft processor design.

IV. Experimental Approach

We have developed a complete flow which implements the stages shown in Fig. 2, including a template for the obfuscation hardware which can be updated on a per-application basis. Modified RTL versions of a three-stage SPREE processor [11] are used to implement the base processor. The obfuscation hardware RTL is generated using scripts based on control flow information collected from code flattening and a user-specified obfuscation function input. Quartus II is used to map modified processor designs to a Stratix IV EP4SGX230 located on an Altera DE4 board. The branching function and lookup table hardware shown in Fig. 3 are isolated in a specific set portion of the FPGA (Fig. 4). If a single soft processor is intended to support numerous obscured applications, the obfuscation hardware can be isolated to a specific region and reconfigured as needed. Both Xilinx and Altera support partial run-time reconfiguration of regions of contemporary FPGAs. Experiments with partial reconfiguration are left as future work. The constraints needed to guide the Quartus II compiler

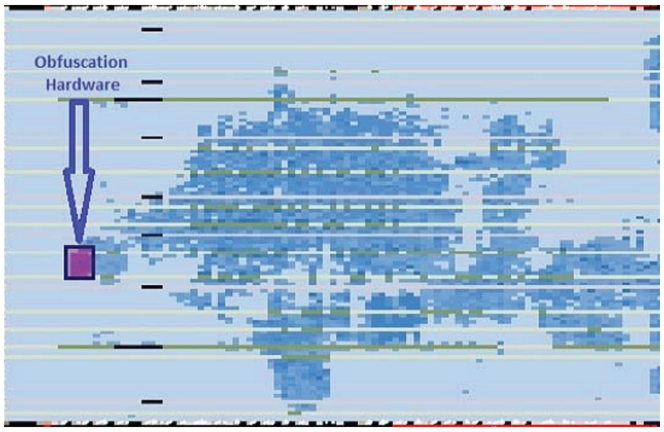


Fig. 4. Isolation of obfuscation hardware in a Stratix IV FPGA

to meet placement objectives can be used once the obfuscation hardware RTL has been created.

The control flow flattening in our flow is performed using aCob software[1]. This tool converts subroutines into the format shown in Fig. 1(b). Our software then converts the $switch$ statement in the flattened code into the series of basic blocks and subroutine calls shown in Fig. 1(c). The SPREE distribution provides a modified MIPS GCC compiler which was used to generate application binaries for the customized processors. Validation of our system performance was performed both using the DE4 board and via ModelSim. Results were validated both for soft processors which fetch instructions from external DRAM and for processors which fetch instructions from internal FPGA SRAM (for proof-of-concept testing). Three benchmarks from the SPREE distribution were used for testing: *quant*, *qsort*, and *crc*.

V. Results

As mentioned in Section I, code obfuscation is used to hide the execution flow of a program, making it difficult for an attacker to identify the details of the algorithm (and especially the conditional statements) which have been implemented. Generally, effective code obfuscation results in a significantly increased number of *potential* execution paths, even if in practice only a smaller set are used. Since, unlike software-only obfuscation, our approach hides the branch decision logic (e.g. the branching function) in hardware which cannot be easily examined by an attacker, the difficulty of determining the control of execution paths is even further obscured. The recent increased use of FPGA soft processors in a variety of embedded and secure systems [12] increases the importance of code obfuscation.

Table I indicates that our added hardware for hardware-assisted code obfuscation is modest, a few hundred lookup-tables (LUTs). The similar size of the obfuscation hardware for all designs reinforces the idea of isolating this hardware in a fixed region. The results include the storage for the lookup table in Fig. 3 which is implemented in FPGA LUTs and flip flops (FFs). As noted in Section I, code obfuscation is a *complementary* security approach to encryption algorithms such as AES, which in contrast, typically require 500-1,000 LUTs [13]. Encryption hides the *values* of instruction and data while code obfuscation hides *control flow* information. The use of code obfuscation to enhance FPGA system security is not resource limiting if it is used with encryption. The maximimum lookup table size across all benchmarks was 33 entries. Our experiments show that the maximum clock speed of the SPREE processor was reduced from about 160 MHz to about 140 MHz for the obfuscated versions, although this performance loss could likely be overcome with an optimized hardware implementation.

[1]http://sourceforge.net/projects/acob/

TABLE I
FPGA resource utilization for standard SPREE processor and processor augmented with obfuscation hardware. Resources are reported in lookup tables (LUTs) and flip flops (FFs).

	SPREE		Obfuscation HW + SPREE	
Program	LUTs	FFs	LUTs	FFs
quant	744	146	964	173
qsort	744	146	912	173
crc	744	146	867	173

Table II indicates the performance, in SPREE clock cycles, of executing the three applications. Although a significant slowdown for each application is noted, our hardware-assisted obfuscation performs roughly the same as a software-only obfuscator (aCob) and is similar to reported performance results from other obfuscators [3]. The run time overhead for execution from DRAM was less for *quant* (3.6×) and *crc* (3.1×). The size of the generated code for our architecture increased by about 70% on average versus unobfuscated code, similar to the code size increase due to aCob (50%).

Various numerical metrics have been developed to measure increases in program complexity [8] which measure the effectiveness of code obfuscation. Almost all of these metrics attempt to measure the number of possible execution paths through the code. Generally, a linear or polynomial increase in the number of control branch points can lead to an exponential increase in the number of possible paths, making an attacker's job much more difficult. One widely used metric, cyclomatic complexity [3] (also known as the McCabe metric), is defined as [8]:

$$cyclomatic = e - n + 2p \qquad (3)$$

where e is the number of branch points in the code, n is the number of code segments, and p are the number of end points. Effectively, an increase in the number of branches e increases complexity and the number of potential control paths. For most code, a cyclomatic number of 10 or less indicates easy-to-follow code. Table III shows that our approach has increased cyclomatic complexity versus the software-only aCob obfuscator. Additionally, unlike previous approaches, control flow decisions in our approach are made in hardware using the *branching function*, completely opaque to potential attackers.

TABLE II
PERFORMANCE RESULTS FOR UNOBFUSCATED CODE, SOFTWARE-ONLY (ACOB), AND HARDWARE-ASSISTED OBFUSCATION

	Clock cycles			
Program	Unobfuscated	aCob	Hardware-obfusc.	increase
quant	24,548	110,104	112,617	4.6×
qsort	17,790	35,553	37,104	2.1×
crc	108,771	553,590	565,895	5.2×

TABLE III
CYCLOMATIC COMPLEXITY RESULTS FOR UNOBFUSCATED CODE, SOFTWARE-ONLY (ACOB), AND HARDWARE-ASSISTED OBFUSCATION

Program	Unobfuscated	aCob	Hardware-obfuscated
quant	9	41	66
qsort	5	14	23
crc	2	11	20

TABLE IV
KNOT COUNT RESULTS FOR UNOBFUSCATED CODE, SOFTWARE-ONLY (ACOB), AND HARDWARE-ASSISTED OBFUSCATION

Program	Unobfuscated	aCob	Hardware-obfuscated
quant	0	5	11
qsort	0	22	32
crc	0	5	11

TABLE V
COMPILE TIME RESULTS FOR UNOBFUSCATED CODE, SOFTWARE-ONLY (ACOB), AND HARDWARE-ASSISTED OBFUSCATION

Program	Unobfuscated	aCob	Hardware-obfuscated	
	software (ms)	software (ms)	software (ms)	HW (s)
quant	0.97	0.98	0.99	124
qsort	1.03	1.04	1.05	116
crc	0.94	0.96	0.97	130

This benefit is not reflected in the results in the table.

Another well-known code complexity metric is *knot count* [8]. This metric examines the number of control flow *crossings* in a program. For example, each branch can be considered as an edge extending from the branch source address to the target address. If edges of two or more branches cross, a *knot* is formed. In general, code flattening and the insertion of a branching function introduce a large number of knots since control flow is centralized in one or a small number of addresses. Table IV indicates the knot count increase of our approach. The use of the hardware-based branching function increases knot count versus software-only flattening. Finally, we examine the program compile time for the three cases of interest (Table V). Although the compile time of the FPGA-based obfuscation hardware is a concern, its small size limits FPGA compile time to about two minutes. As mentioned in Section IV, this generated hardware bitstream can be combined with the static bitstream for the remainder of the SPREE processor.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have described an FPGA-specific technique to obscure code executed by FPGA soft microprocessors. Our system flow allows designers to specify an obfuscation function which is implemented with a small amount of FPGA hardware. Our software flow generates both the obfuscated binary (after control flattening) and the hardware necessary to perform obfuscation. Our technique is more secure than a software-only approach since specialized branching decisions are made using a branching function embedded within secure FPGA hardware. FPGA reconfigurability allows this hardware to be customized on a per-application basis, based on programmer input. In the future, we will explore combining our approach with dynamic run-time compilation to further obscure program address traces. The development of security metrics to quantify the benefits of hardware-assisted obfuscation is also a direction for future work.

VII. ACKNOWLEDGMENTS

This research was funded by the National Science Foundation under grant CNS-1318497. We thank Altera for providing the DE4 board and Quartus software.

REFERENCES

[1] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” University of Auckland, Technical Report, 1997.

[2] C. Wang, “A security architecture for survivability mechanisms,” Ph.D. dissertation, University of Virginia, Oct. 2000.

[3] T. Laszlo and A. Kiss, “Obfuscating C++ programs via control flow flattening,” *Sectio Computatorica*, Aug. 2009.

[4] A. Majumdar, “Design and evaluation of software obfuscations,” Ph.D. dissertation, Department of Computer Science, University of Auckland, New Zealand, Oct. 2008.

[5] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” in *ACM Conf. on Computer and Comm. Security*, Oct. 2003, pp. 290–299.

[6] D. Aucsmith, “Tamper resistant software: An implementation,” in *International Workshop on Information Hiding*, May 1996, pp. 317–333.

[7] P. Falcarin, R. Scandariato, and M. Baldi, “Remote trust with aspect oriented computing,” in *IEEE Advanced Information and Networking Applications*, Apr. 2006, pp. 317–333.

[8] B. Anckaert, M. Madou, B. D. Sutter, K. D. Bosschere, and B. Preneel, “Program obfuscation: A quantitative approach,” in *ACM Workshop on Quality of Protection*, Oct. 2007, pp. 15–20.

[9] S. Schrittwieser and S. Katzenbeisser, “Code obfuscation against static and dynamic reverse engineering,” in *International Conference on Information Hiding*, 2011, pp. 270–284.

[10] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande, “Hardware assisted control flow obfuscation for embedded processors,” in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Sep. 2004, pp. 292–302.

[11] P. Yiannacouras, J. G. Steffan, and J. Rose, “Application-specific customization of soft processor microarchitecture,” in *Proc. Int’l Symp. on FPGAs*, Feb. 2006, pp. 201–210.

[12] J. Crenne, R. Vaslin, G. Gogniat, J.-P. Diguet, R. Tessier, and D. Unnikrishnan, “Configurable memory security in embedded systems,” *ACM Transactions on Embedded Computer Systems*, vol. 12, no. 3, pp. 1–25, Mar. 2013.

[13] P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvoy, “Implementation of the AES-128 on Virtex-5 FPGAs,” in *Proceedings, AFRICACRYPT*, 2008, pp. 16–26.