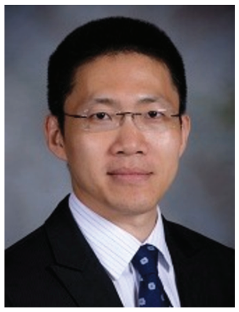
Security by Compilation: An Automated Approach to Comprehensive Side-channel Resistance



Chao Wang
University of Southern California

Patrick Schaumont
Virginia Tech

We explain how formal verification and program synthesis can be used to (1) detect side-channel leaks of software code running on portable devices, (2) prove the absence of side-channel leaks, and (3) transform software to eliminate such leaks. We use power side-channel leaks in cryptographic software as examples, but the underlying techniques are applicable to other types of side channels and software systems as well.

1. INTRODUCTION

Programmers often view the computing devices as blackboxes, but real computers leak information of the software they execute through various side channels, e.g., variations in power dissipation, radiation, execution time, and sound signature of the processor. Side-channel information may be exploited by adversaries. For example, while cryptographic algorithms may be secure against hundreds of years of brute-force attacks, their actual implementations may be broken in hours or even minutes in the presence of side-channel leaks.

Since the seminal work on differential power analysis [Kocher et al. 1999], many similar techniques have been developed, making side-channel analysis (SCA) a real threat to commercial hardware and software. Examples from the past few years include the use of SCA to extract secret keys from contactless smartcards [Kasper et al. 2011], keyless entry systems [Eisenbarth et al. 2008], secure memory modules [Balasch et al. 2012], and field programmable gate arrays (FPGAs) [Moradi et al. 2011; Skorobogatov and Woods 2012; Moradi et al. 2013]. Furthermore, new sources of side-channel leakage and methods to exploit them are discovered on a regular basis. For example, the *sound* made by a processor as it executes the public-key RSA algorithm was found to be a viable side channel to reveal the secret key [Genkin et al. 2014] — although a patch was quickly made and distributed, it is still worrisome that side-channel leakage can be found in such widely distributed production software.

A fundamental difficulty in mitigating side-channel leaks is that their physical sources are at the level of the processor hardware, an abstraction layer typically invisible to programmers. For example, data-dependent control flow in software will impact the power dissipation of the processor hardware, but for an average programmer, it may be difficult or even impossible to tell how much harmful side-channel leakage would occur. A similar problem exists with the prediction of execution time, another source of side-channel leakage, or the prediction of cache misses and their impact on the execution time. Average programmers tend to view the computing device as an abstract machine while ignoring architectural details of the processor (such as out-of-order execution) and storage (such as register allocation and cache), thus making side-channel leakage a concept that is hard to grasp for them.

Current implementation of side-channel resistant software in embedded computing applications relies on manual efforts of experts, but even for them, this process is complex and error-prone. Furthermore, there are no techniques available to verify these handcrafted implementations, let alone generating them automatically. Although automation is desirable, existing verification and synthesis techniques are not sufficient. The reason is because, first, side-channel resistance is a *non-functional* property, which cannot be handled by techniques developed for proving functional correctness [Clarke et al. 1999; McMillan 1994]. Furthermore, unlike the *non-interference* property in information-flow security [Sabelfeld and Myers 2003], side-channel resistance is a *statistical* property, which requires fundamentally new analysis techniques.

We outline the development of a new type of verification and program synthesis techniques to aid in the construction of side-channel resistant software for embedded computing applications, e.g., cryptographic software used in various cyber-physical systems (CPS) and the Internet of things (IoT) where physical security of the computing devices is a major concern. As shown in Fig. 1, the automated analysis and code transformation framework consists of techniques being developed along the following directions:

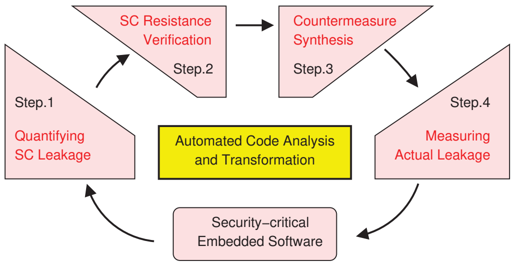

SC Resistance Verification
Step.2
Countermeasure Synthesis
Step.3
Step.1
Quantifying SC Leakage
Automated Code Analysis and Transformation
Step.4
Measuring Actual Leakage
Security–critical Embedded Software

Fig. 1. Automated approach to side-channel resistance.

— *Quantifying side-channel leaks*. First, we need to formally define what it means for a piece of software to be side-channel resistant on a given platform, and in case it is not side-channel resistant, how to quantify the amount of leakage.
— *Verifying side-channel resistance*. For existing and manually-secured software code, we need new verification techniques to formally prove that the implementation is indeed side-channel resistant.
— *Synthesizing countermeasures*. We also need program synthesis techniques for automatically generating functionally-equivalent, but side-channel resistant, software code to replace the original code. They must go beyond simple compiler transformations, to handle unknown vulnerabilities and generate new implementations.
— *Validation on real devices*. Finally, the resulting software code must be validated on real devices to ensure our modeling and synthesis accurately reflect side-channel leaks observed in the physical world.

In the remainder of this article, we use power side-channel leaks in cryptographic software as examples to illustrate our recent work on formally verifying side-channel resistance [Eldib et al. 2014b; Eldib et al. 2014c] as well as synthesizing countermeasures [Eldib and Wang 2014b]. Then, we discuss how to extend these techniques to handle other types of side channels and software systems.

2. PRELIMINARIES

We assume the software code implements a cryptographic function $c \leftarrow f(x, k)$, where x is the plaintext, c is the ciphertext, and k is the secret key. The goal of the adversary is to compute k based on knowledge of x and c as well as the information of internal computations leaked through side channels.

It is possible to implement $c \leftarrow f(x, k)$ in a manner such that the side-channel leakage remains harmless, e.g., using the idea of secret sharing [Chari et al. 1999]. In this approach, every internal variable v of the software program is split into $n + 1$ shares $v_0, v_2, \ldots, v_n$ such that $v = v_0 \oplus v_1 \oplus \ldots \oplus v_n$, where $\oplus$ is a suitable masking operator, e.g., the XOR operator in Boolean domain. Among these $n + 1$ shares, n are randomly chosen masks and the remaining one is computed as a matching share. Since every masked share v_i is statistically independent of the original v, leakage of individual shares or any combination of $\leq n$ shares will not reveal v.

Splitting variables into shares affects the internal operations of the program. Thus, we call the new program a *masked* program. Furthermore, the number of shares corresponds to the *order* of masking, e.g., in an *order*-d masking, every variable is split into $d + 1$ shares. If $f(x, k)$ were a linear function of k with respect to XOR, masking would be straightforward, because $f(x, k \oplus r) \oplus f(x, r) = f(x, k) \oplus f(x, r) \oplus f(x, r) = f(x, k)$. That is, we can mask the sensitive k by the XOR with a random variable r before the computation, and de-masking afterward by the XOR with $f(x, r)$. However, in practice, $f(x, k)$ is always a non-linear function, which means masking requires a complete rewriting of the software code, and the process is labor-intensive and error-prone.

Threat Model. We assume an adversary knows the value of the plaintext x, the ciphertext c, and side-channel information of at most d intermediate computation results; they correspond to variables in the program. Let $I_1, I_2, \ldots, I_d$ be the set of intermediate results. Furthermore, each $I_i(x, k, r)$ is a function in terms of x, k, and random variable r introduced to *mask* the sensitive k. Thus, the adversary does not have access to the value of r. However, if the side-channel leakage associated with I_i or any combination of $\leq d$ intermediate results is dependent of k, we say the implementation of $c \leftarrow f(x, k)$ is vulnerable to SCA based attacks.

A necessary condition for $f(x, k)$ to be side-channel resistant is that all intermediate computation results are either logically independent of k or logically dependent of (and thus masked by) some random variable r. The condition seems reasonable and can be easily checked [Bayrak et al. 2013]. However, it is a logical property (as opposed to statistical property)—we will show that the condition is not sufficient for ensuring side-channel resistance.

Leakage Model. A widely used power model is the *Hamming Weight (HW)* model, which relates variations in the power dissipation of the processor to values of its registers, which in turn hold variables used in the software program. More specifically, the power dissipation correlates to the number of logical-1 bits of intermediate computation results. We have shown in our work [Eldib et al. 2014c] that the HW model is sufficiently accurate for conducting DPA attacks on embedded systems. Sometimes, however, the *Hamming Distance (HD)* model needs to be used instead, to relate variations in power dissipation to differences between the register values and their initial states [Brier et al. 2004].

The example in Fig. 2 shows that, under the HW model, *logically dependent of some random variable* is not the same as *statistically-independent of the secret*. Here, `k` is the secret bit, `r1` and `r2` are the random bits, and `o1, o2, o3`, and `o4` are the masked intermediate results. According to the truth table on the right-hand side or functions on the left-hand side, all four intermediate results are logically dependent of `r1,r2` and thus are masked. However, the first three still leak secret information because they are not perfectly masked. Specifically, `o1` leaks information of `k` because, if it were logical 1, `k` would also be logical 1 regardless of the values of the random variables. `o2` leaks information of `k` because, if it were logical 0, `k` would also be logical 0. `o3` leaks information of `k` because, if it were logical 1 (or 0), there would be a 75% chance that `k`

```
o1 = x ∧ k ∧ (r1 ∧ r2)
o2 = x ∧ k ∨ (r1 ∧ r2)
o3 = x ∧ k ⊕ (r1 ∧ r2)
o4 = x ∧ k ⊕ (r1 ⊕ r2)
```

x	k	r1	r2	o1	o2	o3	o4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	1

Fig. 2. Four masking schemes with different side-channel leakages and the corresponding truth table when $x = 0$. Although `o1,o2,o3` are masked by random bits `r1` and `r2`, they still leak secret information about `k`. In contrast, `o4` does not have side-channel leakage.

is also logical 1 (or 0). In contrast, `o4` does not leak information of `k` because, regardless of whether `k` is logical 1 (or 0), there is a 50% chance that `o4` is logical 1 (or 0).

Perfect Masking. Following [Blömer et al. 2004], we define *perfect masking* for the implementation of $c \leftarrow f(x, k)$ as follows. Given a pair (x, k) of plaintext and secret key, together with d intermediate results $I_1(x, k, r), \ldots, I_d(x, k, r)$, where r is a random variable in the domain R, we say f is order-d *perfectly masked* if the joint distribution of $I_1, \ldots, I_d$ is independent of k. Otherwise, we say the implementation is vulnerable to *order*-d SCA-based attacks. The intermediate result `o4` in Fig. 2, for example, is perfectly masked and thus is immune to first-order attacks.

3. VERIFYING THE SIDE-CHANNEL RESISTANCE

A verification procedure for deciding *if* $f(x, k)$ *is perfectly masked* works as follows. Initially, the input variables are annotated such that all plaintext bits in x are marked as public, all key bits in k are marked as secret, and all bits in r are marked as random. Then, for each intermediate result, denoted $I(x, k, r)$, the procedure checks if I is perfectly masked by r.

For ease of presentation, we assume $d = 1$. Thus, verifying if I is perfectly masked is the same as checking the *validity* of the following formula:

$$\forall x. \forall k. \forall k'. \left(\sum_{r \in R} I(x, k, r) = \sum_{r \in R} I(x, k', r) \right)$$

Here, x denotes the plaintext value, k and k' denote two values of the key, and r denotes the random variable. Thus, for each combination (x, k, k'),

— $\sum_{r \in R} I(x, k, r)$ denotes the number of values of r making I logical 1; and
— $\sum_{r \in R} I(x, k', r)$ denotes the number of values of r making I logical 1.

Assume that r is uniformly distributed in R, the above summations are probabilities of I being logical 1 under the plaintext value x and the two key values k and k'.

Given a cryptographic software program, we first obtain a branch-free representation by merging all if-else branches. Since there are typically no input-dependent loop bounds (otherwise, they may be timing side channels), we apply loop unrolling to obtain a loop-free program. Since all variables are bounded integers, we can model them as finite-length bit-vectors or even construct a purely Boolean program. Fig. 3 shows a masked version of $c \leftarrow (k1 \wedge k2)$, where $r1$ and $r2$ are two random bits. The corresponding de-masking function, which is not shown in the figure, would be $c \oplus (r1 \wedge r2)$. Due to the property of XOR, de-masking would produce the desired value $(k1 \wedge k2)$.

Thus, we can traverse the abstract syntax tree (AST) of the given program, and for each intermediate result I, check if I is perfectly masked. For ease of implementation, instead of checking the validity of the above universally-quantified formula, we use a

```
compute(bool k1, bool k2, bool r1, bool r2){
  bool n1, n2, n3, n4, n5, n6, n7, n8, c;
  n1 = k1 ⊕ r1;
  n2 = k2 ⊕ r2;
  n3 = n1 ∧ n2;
  n4 = k2 ⊕ r2;
  n5 = r1 ∧ n4;
  n6 = k1 ⊕ r1;
  n7 = r2 ∧ n6;
  n8 = n5 ⊕ n7;
  c = n3 ⊕ n8;
  return c;
}
```

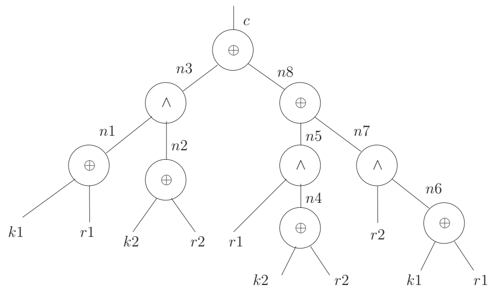


Fig. 3. Example Boolean program and its graphic representation (⊕ denotes XOR; ∧ denotes AND).

constraint solver to check the satisfiability of its negation, shown as follows:

$$\exists x.\exists k.\exists k' \,.\, \left(\sum_{r \in R} I(x,k,r) \neq \sum_{r \in R} I(x,k',r) \right)$$

If the formula is satisfiable, the solver will return a plaintext value x and two different key values (k, k') such that the probabilities of $I(x, k, r)$ and $I(x, k', r)$ being logical 1 differ. Therefore, some information of k is leaked. In contrast, if this formula is unsatisfiable, it means no such leak is possible.

Model Counting. Thus, the verification of side-channel resistance can be viewed as comparing the number of satisfying assignments of two closely-related formulas. This is the case not only for power side channels, but also for other types of side channels because, fundamentally, the attacks all rely on correlation-based statistic analysis. Consequently, unlike standard verification techniques, which rely on SAT and SMT solvers as the decision procedures, the new verification techniques need SAT$^{\#}$ and SMT$^{\#}$ solvers to support model-counting. Although model-counting solvers are not yet as mature as standard SAT and SMT solvers in terms of speed and scalability, they are catching up rapidly [Chakraborty et al. 2013; Chakraborty et al. 2014; Aydin et al. 2015; Fremont et al. 2017].

Without using specialized solvers, we can still solve the verification problem [Eldib et al. 2014b], albeit in a less efficient fashion. Fig. 4 is a pictorial illustration of our encoding for $I(k_1, k_2, r_1, r_2)$, where k_1, k_2 are key bits and r_1, r_2 are random bits. Each box in the figure denotes a copy of the input-output relation of I but with random bits customized to values 00, 01, 10, and 11, respectively. Furthermore, the first four boxes correspond to one set of key values, denoted k_1 and k_2, and the remaining four boxes correspond to another set of key values, denoted k'_1 and k'_2. The summations add up the number of logical 1's,

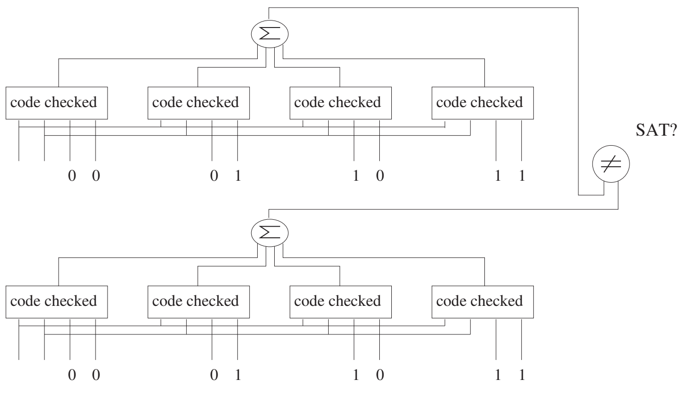


Fig. 4. Checking the statistical dependence of secret data $(k1, k2)$.

while the comparison on the right-hand side checks if the probabilities of $I(k_1, k_2, \ldots)$ and $I(k'_1, k'_2, \ldots)$ being logical 1 can differ.

Compositional Verification. Whether we use specialized solvers or standard solvers does not change the fact that, in the worst case, the number of satisfying assignments is exponential in the number of random bits in r. This may cause scalability problems. Fortunately, certain properties of masked programs allow us to apply compositional analysis. That is, instead of verifying the whole program, we partition the AST into small code regions, and apply the model-counting based analysis only to each individual code region, one at a time.

This is possible because a common strategy used by cryptographic system engineers is to create a chain of small modules, where the inputs of each module are masked before executing its logic and are de-masked afterward. To avoid having unmasked intermediate values, the inputs to the successor module are masked with fresh random variables before they are de-masked from the random variables used by the previous module. Due to the *associativity* of XOR ($\oplus$), reordering these masking and de-masking operations would not change the result. We have shown [Eldib et al. 2014b] that such property may be exploited for performance optimization in real applications.

Quantifying the Leakage. Our verification procedure so far only checks if a given program is perfectly masked. However, it cannot quantify the amount of leakage in programs that are not perfectly masked. To differentiate the strengths of masking schemes, e.g., `o1,o2,o3` in Fig. 2, we have extended the definition of perfect masking to quantify the amount of residual leakage [Eldib et al. 2014c]. That is, we define the *quantitative masking strength (QMS)* as the minimal value of $(1 - \Delta_{qms})$ such that,

$$|E(I_i \mid k = \kappa \wedge x = \chi) - E(I_i \mid k = \kappa' \wedge x = \chi)| \leq \Delta_{qms}$$

holds for all intermediate results $I_i(x, k, r)$, all plaintext values χ, and all key values κ and κ', where $\kappa \neq \kappa'$. Here, $E(I_i \mid k = \kappa \wedge x = \chi)$ can be viewed as the number of values of r under which $I_i(\kappa, \chi, r)$ evaluates to logical 1.

Consider the example in Fig. 2 again. We have

$$\begin{array}{ll}
\Delta_{qms}(o1) = 1/4 - 0/4 = 0.25 & \Delta_{qms}(\overline{o1}) = 4/4 - 3/4 = 0.25 \\
\Delta_{qms}(o2) = 4/4 - 1/4 = 0.75 & \Delta_{qms}(\overline{o2}) = 3/4 - 0/4 = 0.75 \\
\Delta_{qms}(o3) = 3/4 - 1/4 = 0.50 & \Delta_{qms}(\overline{o3}) = 3/4 - 1/4 = 0.50 \\
\Delta_{qms}(o4) = 2/4 - 2/4 = 0.00 & \Delta_{qms}(\overline{o4}) = 2/4 - 2/4 = 0.00
\end{array}$$

Intuitively, the numbers are consistent with the amount of power leakage. For example, the perfectly-masked `o4` has $\Delta_{qms} = 0.0$, which corresponds to $QMS = 1.0$. For each of the remaining three, the larger Δ_{qms}, the more information it leaks.

To decide whether a given software program meets the QMS requirement, we check if there exists any intermediate result $I(x, k, r)$ that satisfies the following formula:

$$\exists x, k, k' \,.\, \left(\sum_{r \in R} I(x, k, r) - \sum_{r \in R} I(x, k', r) \right) > \Delta_{qms} \,.$$

If this formula is satisfiable, there exist some values for x and (k, k') such that the difference between distributions of $I(x, k, r)$ and $I(x, k', r)$ is larger than the expected Δ_{qms}. On the other hand, if the above formula is unsatisfiable for all intermediate results of the program, the implementation meets the QMS requirement.

4. SYNTHESIZING SIDE-CHANNEL RESISTANT SOFTWARE

Given some not-yet-masked software code as input, we use *inductive program synthesis* to systematically search for an alternative, functionally-equivalent, but side-channel resistant implementation. Although recent years have seen a renewed interest

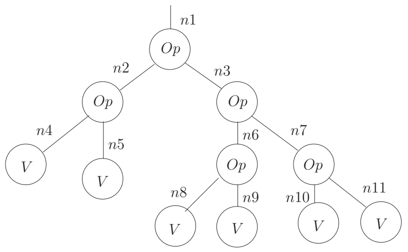


Fig. 6. A candidate program skeleton consisting of 11 parameterized AST nodes.

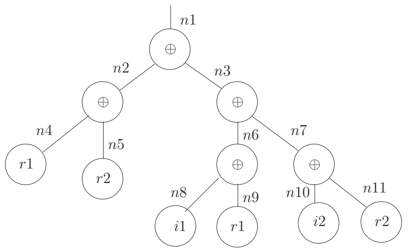


Fig. 7. The synthesized candidate program with instantiated Boolean masking.

in applying *inductive synthesis* to a wide variety of applications [Solar-Lezama et al. 2005; Jha et al. 2010; Gulwani 2011; Harris and Gulwani 2011; Harris et al. 2013; Alur et al. 2013; Eldib and Wang 2014a; Eldib et al. 2016], prior to our work, it has never been used to mitigate side-channel leaks.

Our synthesis procedure relies on a set of architectural parameters to estimate the leakage. For each side channel, we leverage a different type of code transformation, or countermeasure, to eliminate the leakage. Specifically, for instruction timing, the countermeasure would be CFG-balancing, which is to remove all branching conditions that are dependent on the sensitive data. For cache-memory timing, the countermeasure would be to remove the dependency between table lookups and the sensitive table content. For power side channel, the countermeasure would be masking, which removes the dependency between variations in power dissipation and the sensitive data.

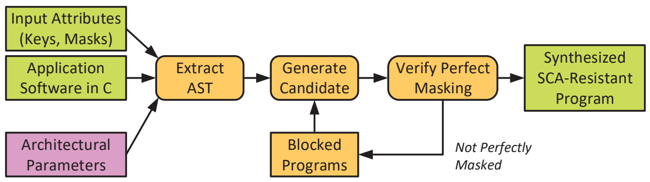


Fig. 5. Counterexample-guided inductive synthesis procedure.

The overall flow of our synthesis procedure is shown in Fig. 5. Given the application software together with a set of sensitive variables and architectural parameters, it first extracts an abstract syntax tree (AST) representation of the program. Then, it generates a *candidate program* that is functionally equivalent to the original program—the two programs produce the same output for the same input. Next, it verifies that the candidate program is free of side-channel leaks. If the verification succeeds, we are done. Otherwise, we block this candidate program and try again.

To generate the candidate program, we create a *skeleton* of the program's AST, which captures any syntactically correct program up to that size. For example, the skeleton AST of size 5 shown in Fig. 6 can represent any candidate program with up to five AST nodes: *Op* represents any of the predefined binary operators, $V|C$ means the node represents either a variable or a constant, and the root node represents the computation output, which must be functionally equivalent to the original program.

We use SMT solvers to search among the candidate programs. That is, to determine the node types, variable names, and constant values of the skeleton AST, we construct a formula Φ such that Φ is satisfiable if and only if the candidate program is functionally equivalent to the original program. If Φ is unsatisfiable, it means no solution exists; in this case, we increase the skeleton size and try again.

If Φ is satisfiable, we have found a candidate program, e.g., as in Fig. 7, which is an instantiation of the skeleton AST. The next step is to verify that it is free of side-channel leaks. Toward this end, we create another formula Ψ such that Ψ is satisfiable if and only if the candidate program has side-channel leaks. If Ψ is unsatisfiable, the candidate program is proved to be a valid solution and we are done. Otherwise, we block this candidate program and try again.

Fig. 8 shows a masked implementation of the χ-function of a reference implementation of MAC-Keccak, which is NIST's new SHA-3 crypto-hashing algorithm [NIST 2013]. The original code is on the left-hand side and the new code is on the right-hand side. We guarantee that all intermediate results in the new program are perfectly masked. That is, by assuming `r1`, `r2` and `r3` are uniformly distributed random variables, our method guarantees that the probability of each intermediate result being logical 1 (or 0) is independent of `i1`, `i2` and `i3`. As for the compactness of the implementation, we note that a countermeasure handcrafted by cryptographic experts has 14 operations [Bertoni et al. 2013], whereas our synthesized version only has 12 operations—it is more compact than the one handcrafted by experts.

```
Chi(bool i1, bool i2, bool i3) {
  bool n1, n2, n3;
  n3 = ¬i2;
  n2 = n3 ∧ i3;
  n1 = n2 ⊕ i1;
  return n1;
}
```

i1	i2	i3	n3	n2	n1
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	1	0	0	1

```
mChi(bool i1, bool i2, bool i3) {
  bool r1, r2, r3;  //random bits added
  bool b1, b2, b3, n1, n2, n3, n4, n5, n6, n7, n8, n9;
  b1 = i1 ⊕ r1;
  b2 = i2 ⊕ r2;
  b3 = i3 ⊕ r3;
  n9 = b3 ∧ r2;
  n8 = r3 ∧ r2;
  n7 = r3 ∨ b2;
  n6 = r1 ⊕ n9;
  n5 = n7 ⊕ n8;
  n4 = b2 ∨ b3;
  n3 = n5 ⊕ n6;
  n2 = n4 ⊕ b1;
  n1 = n2 ⊕ n3;
  return n1;
}
```

Fig. 8. The χ function in MAC-Keccak, its truth-table, and the synthesized $m\chi$ function ($\neg$ denotes NOT, $\wedge$ denotes AND, $\vee$ denotes OR, and $\oplus$ denotes XOR).

Compositional Synthesis. Again, the key is to exploit the unique characteristics of masked programs. Thus, we have developed a compositional synthesis procedure [Eldib and Wang 2014b], which applies computationally intensive analysis (e.g., model-counting) only to small code regions, one at a time, as opposed to the entire program. Compared to the application of standard synthesis techniques to the entire program, our compositional synthesis procedure is significantly more scalable.

5. VALIDATING SIDE-CHANNEL RESISTANCE ON REAL DEVICES

To confirm that our modeling and analysis of side-channel leaks at the source code level accurately reflect what is observed in the physical world, we conducted a set of SCA-based attacks on implementations of MAC-Keccak, AES, and a few other cryptographic algorithms [Eldib et al. 2015]. In these experiments, we ran all software code on a 32-bit Microblaze processor [Xilinx 2014] built on a Xilinx Spartan-3e FPGA (Fig. 9). To measure the power dissipation of the processor core, we used a Tektronix DPO 3034 oscilloscope and a CT-2 current probe to sample the power dissipation. The side-channel attack shown in Fig. 9 was conducted using the classic differential power analysis, i.e., difference of means [Kocher et al. 1999]. To limit the effect of measurement noise, we collected each *trace* after running the same software code 128 times and using the oscilloscope to calculate the average. Thus, a trace refers to a set of samples taken during the execution of the software code.

We used differential power analysis (DPA) to determine if a key guess was correct. Recall that DPA relies on the observation that power dissipation variations correlate to the values of the sensitive bits being manipulated. Using the same input vector stream of plaintext as in the measured traces, we computed the value of the sensitive variable assuming that the secret key was one of the key guesses. For an n-bit key, there would be 2^n key guesses. For each key guess, we divided the set of measurement traces into two bins, one for all the sensitive values of logic 0, and one for all the sensitive values of logic 1. Then, we computed the difference of means between those two bins for each key guess, and selected the key guess that result in the maximum difference.

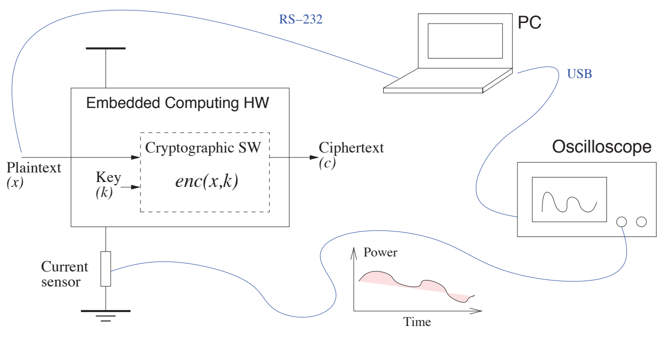


Fig. 9. The power side-channel attack system setup.

Fig. 10 shows our results on the SHA3 benchmark. The x-axis denotes the QMS value as defined in Section 3, while the y-axis (in logarithmic scale) denotes the number of traces needed to determine the secret key. In addition to the measured data, which are the stars in the figure, we plotted an empirical approximation rule (dotted curve) generated by hit-and-trial to estimate the measured data. We can see that when the QMS approaches 1.0, the number of traces needed to determine the secret key approaches infinity. However, when the QMS deviates from 1.0 slightly, the number of traces needed to determine the secret key drops quickly. Overall, the side-channel resistance as measured by the number of traces needed to determine the secret key is dependent on QMS. Fig. 11 shows our results on the AES benchmark.

In both cases, the approximate empirical formula computed to estimate the number of required DPA traces has the following relation with the QMS value:

$$N_{trace} = \frac{1}{(1 - \text{QMS})^c} \ ,$$

where $c \approx 2.0$. Note that we obtained this equation without prior knowledge of what the relation should look like. Later, we discovered that it matches the theoretical analysis result in the literature [Mangard 2004], which says that c should be precisely 2.0 as opposed to ≈ 2.0, since $(1 - \text{QMS})$ represents the standard deviation of power analysis measurements.

6. FUTURE DIRECTIONS

The next step is to generalize the verification and program synthesis techniques to handle other types of side channels and software systems. We envision a comprehensive framework (Fig. 12) whose input is the source code of some security-critical software, together with a set of sensitive variables (keys, passwords, etc.) tagged in the source code. To support modeling of various types of side-channel leakage, it also accepts a set of architecture parameters and leakage models. The output is a transformed

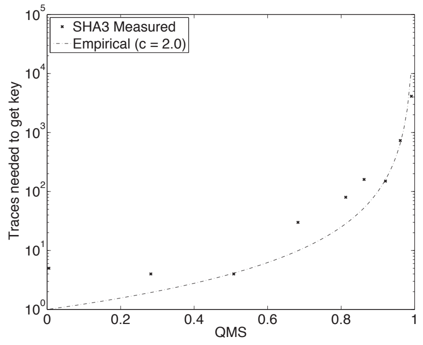


Fig. 10. DPA attacks on MAC-Keccak: number of traces needed versus the QMS.

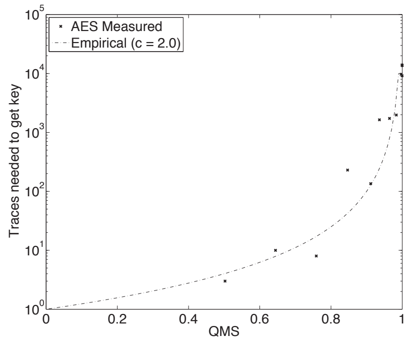


Fig. 11. DPA attacks on AES: the number of traces needed versus the QMS value.

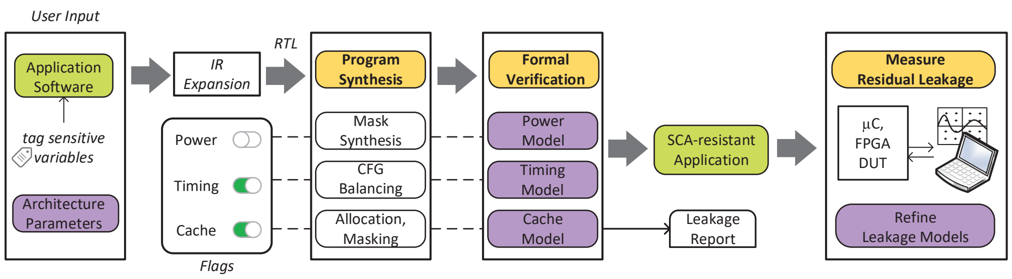


Fig. 12. Framework for synthesizing, verifying, and validating side-channel countermeasures. The input includes the source code of an application, a list of sensitive variables, and the parametric architecture definition. The compiler-like tool can (a) insert countermeasures through inductive synthesis and (b) statically detect remaining side-channel leakage in either synthesized or manually programmed countermeasures.

application for which the dependency between side-channel leakage and sensitive variables is removed.

When a programmer develops an application, for example, he or she will indicate one or more types of side-channel leakage, including power dissipation, instruction time, and cache-memory timing behavior. The framework will examine the software code to check for the presence of side-channel leakage. In the presence of side-channel leakage, the framework will leverage inductive synthesis to transform the software code into an implementation that eliminates the side-channel leakage. The framework also assesses the quality of the countermeasure. In addition, by measuring and analyzing the actual leakage of driver applications using a hardware prototype, we will refine the architecture parameters and leakage models, thus improving verification and countermeasure synthesis.

Advantages over Alternative Approaches. There are significant efforts on eliminating physical emissions of sensitive equipments and electronic systems, e.g. in the legendary TEMPEST project [TEMPEST 1972]. There are also techniques for reducing physical emissions of hardware (microcontrollers, FPGAs, ASICS, and CPUs) – although these techniques are theoretically feasible, they are not economical. In contrast, our approach does not aim to eliminate physical emissions of the computing devices; instead, it transforms the software running on these devices to make the computation *leak-resistant*. Therefore, our approach is fundamentally more economical and thus more widely applicable.

Another alternative is the use of side-channel resistant software libraries developed by experts. While libraries could help for selected cases of reusable functionality on some widely deployed platforms, it is not scalable in general, for several reasons. First, since side-channel leakage is platform-specific, side-channel resistant libraries must also be platform-specific and thus non-portable. Second, side-channel resistant techniques incur performance penalty, which means an expert has to decide which sources of side-channel leakage to address and what level of residual leakage should be tolerated. Therefore, a *universal* countermeasure library is not meaningful; in practice, the application context is crucial to decide on what makes sense and what not.

7. RELATED WORK

Formal Verification. We started with the notion of *perfect masking* introduced by [Blömer et al. 2004] and developed *SC-Sniffer* [Eldib et al. 2014a; 2014b], the first automated tool for formally verifying that a software program is perfectly masked. In comparison, the *Sleuth* tool developed by [Bayrak et al. 2013] can only check if sensitive data are masked by some random variables (a logical property), but cannot check if the masking is perfect (a statistical property). We also extended the notion of perfect masking to quantify the amount of residual leakage in software that are not perfectly masked [Eldib et al. 2014c]. The strength of masking may be computed statically on the source code of the software program, and its accuracy as an indicator for side-channel resistance has been validated by DPA attacks on real devices [Eldib et al. 2015].

Countermeasure Synthesis. There is a large body of work on masking countermeasures for cryptographic algorithms [Messerges 2000; Goubin 2001; Oswald et al. 2005; Herbst et al. 2006; Canright and Batina 2008; Moradi et al. 2011; Barthe et al. 2016], but they require manual design and implementation. By leveraging our verification procedure for proving side-channel resistance, we developed *SC-Masker* [Eldib and Wang 2014b], a tool for automatically synthesizing perfectly-masked software code. Although there exist some other compiler-like tools for mitigating side-channel leaks [Bayrak et al. 2011; Moss et al. 2012; Agosta et al. 2012], they rely on *ad hoc* techniques, e.g., matching some code patterns and applying predefined transformations, as opposed to inductive program synthesis techniques. The main advantage of using inductive synthesis is that the tool becomes application-agnostic and it no longer relies on existing patterns or mitigation strategies. Therefore, it can handle unknown and unexpected vulnerabilities.

Other Side Channels. Besides power side channels, there are other types of side channels through which sensitive information may be leaked. They include, for example, instruction timing side channels [Kocher 1996; Köpf and Dürmuth 2009], cache timing side channels [Grabher et al. 2007], string-related side channels [Bang et al. 2016], and fault-related side channels [Biham and Shamir 1997]. In addition to CPUs, side channels have been identified in GPUs [Jiang et al. 2016; Luo et al. 2015]. Techniques for mitigating some of these side-channel leaks are also proposed. For example, Köpf *et al.* developed techniques for quantitative information flow analysis [Köpf et al. 2012; Backes et al. 2009]. Doychev *et al.* [Doychev et al. 2013] developed static analysis techniques for detecting leaks through cache side channels. Barthe *et al.* [Barthe et al. 2014] developed techniques for mitigating concurrent cache attacks.

8. CONCLUSIONS

We have presented an automated approach to comprehensive side-channel resistance for embedded computing applications. It relies on formal verification techniques to detect side-channel leaks or prove that leaks do not exist, and program synthesis techniques to generate secure implementations. It also leverages hardware prototyping to validate the effectiveness of these verification and synthesis techniques. Although we have used power side-channel leaks in cryptographic software as examples, the underlying techniques may be applied to various side channels in a wide range of embedded processing systems, e.g., in phones, cars, and home appliances, as well as industrial, medical, and transportation systems.

ACKNOWLEDGMENTS

This work was primarily supported by NSF under grants CNS-1617203, CNS-1128903 and CNS-1115839. Partial support was provided by ONR under grant N00014-13-1-0527. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *ACM/IEEE Design Automation Conference*. 77–82.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design*. 1–17.

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*. 255–272.

Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*. 141–153.

Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. 2012. Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs. In *CT-RSA: The Cryptographers' Track at the RSA Conference*. 19–34.

Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 193–204.

Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong non-interference and type-directed higher-order masking. In *ACM SIGSAC Conference on Computer and Communications Security*. 116–129.

Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. Leakage resilience against concurrent cache attacks. In *International Conference on Principles of Security and Trust*. 140–158.

Ali Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. 2013. Sleuth: Automated Verification of Software Power Analysis Countermeasures. In *Workshop on Cryptographic Hardware and Embedded Systems*. 293–310.

Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the Design Automation Conference*. 230–235.

Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. 2013. Keccak implementation overview. URL: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf. (2013).

Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *International Cryptology Conference*. 513–525.

Johannes Blömer, Jorge Guajardo, and Volker Krummel. 2004. Provably secure masking of AES. In *Selected Areas in Cryptography*. 69–83.

Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *Workshop on Cryptographic Hardware and Embedded Systems*. 16–29.

David Canright and Lejla Batina. 2008. A very compact "perfectly masked" S-Box for AES. In *International Conference on Applied Cryptography and Network Security*. 446–459.

Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-aware sampling and weighted model counting for SAT. In *AAAI Conference on Artificial Intelligence*. 1722–1730.

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*. 200–216.

Suresh Chari, Charanjit Jutla, Josyula Rao, and Pankaj Rohatgi. 1999. Towards sound approaches to counteract power-analysis attacks. In *International Cryptology Conference*. 398–412.

E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA.

Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*. 431–446.

Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. 2008. On the power of power analysis in the real world: A complete break of the KeeLoqCode Hopping scheme. In *International Cryptography Conference*. 203–220.

Hassan Eldib and Chao Wang. 2014a. An SMT based method for optimizing arithmetic computations in embedded software code. *IEEE Trans. on CAD of Integrated Circuits and Systems* 33, 11 (2014), 1611–1622.

Hassan Eldib and Chao Wang. 2014b. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*. 114–130.

Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014a. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 11:1–24.

Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014b. SMT based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 62–77.

Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014c. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*. 209:1–6.

Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2015. Quantitative masking strength: Quantifying the side-channel resistance of masked software code. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 34. 10:1558–1568.

Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *International Conference on Computer Aided Verification*. 343–363.

Daniel J. Fremont, Markus N. Rabe, and Sanjit A. Seshia. 2017. Maximum model counting. In *AAAI Conference on Artificial Intelligence*. 3885–3892.

Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*. 444–461.

Louis Goubin. 2001. A sound method for switching between boolean and arithmetic masking. In *Workshop on Cryptographic Hardware and Embedded Systems*. 3–15.

Philipp Grabher, Johann Großschädl, and Dan Page. 2007. Cryptographic side-channels from low-power cache memory. In *International Conference on Cryptography and Coding*. 170–184.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 317–330.

William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 317–328.

William R. Harris, Somesh Jha, Thomas W. Reps, Jonathan Anderson, and Robert N. M. Watson. 2013. Declarative, temporal, and practical programming with capabilities. In *IEEE Symposium on Security and Privacy*. 18–32.

Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. 2006. An AES smart card implementation resistant to power analysis attacks. In *International Conference on Applied Cryptography and Network Security*. 239–252.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*. 215–224.

Zhen Hang Jiang, Yunsi Fei, and David R. Kaeli. 2016. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High Performance Computer Architecture*. 394–405.

Timo Kasper, David Oswald, and Christof Paar. 2011. Side-channel analysis of cryptographic RFIDs with analog demodulation. In *RFIDSec*. 61–77.

Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *International Cryptology Conference*. 104–113.

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *International Cryptology Conference*. 388–397.

Boris Köpf and Markus Dürmuth. 2009. A provably secure and efficient countermeasure against timing attacks. In *IEEE Symposium on Computer Security Foundations*. 324–335.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification*. 564–580.

Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David R. Kaeli. 2015. Side-channel power analysis of a GPU AES implementation. In *IEEE International Conference on Computer Design*. 281–288.

Stefan Mangard. 2004. Hardware countermeasures against DPA – A statistical analysis of their effectiveness. In *The Cryptographers' Track at the RSA Conference 2004*. 222–235.

K. L. McMillan. 1994. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA.

Thomas S. Messerges. 2000. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption*. 150–164.

Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In *ACM Conference on Computer and Communications Security*. 111–124.

Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. 2013. Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: facilitating black-box analysis using software reverse-engineering. In *FPGA*. 91–100.

Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. 2011. Pushing the limits: A very compact and a threshold implementation of AES. In *EUROCRYPT*. 69–88.

Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *Workshop on Cryptographic Hardware and Embedded Systems*. 58–75.

NIST. 2013. Keccak reference code submission to NIST's SHA-3 competition (Round 3). URL: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip. (2013).

Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. 2005. A side-channel analysis resistant description of the AES S-Box. In *International Workshop on Fast Software Encryption*. 413–423.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

Sergei Skorobogatov and Christopher Woods. 2012. Breakthrough silicon scanning discovers backdoor in military chip. In *Workshop on Cryptographic Hardware and Embedded Systems*. 23–40.

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 281–294.

TEMPEST. 1972. A signal problem – the story of the discovery of various compromising radiations from communications and Comsec equipment. *Cryptologic Spectrum, Vol. 2, No. 3, National Security Agency, partially FOAI declassified 2007-09-27* (1972).

Xilinx. 2014. MicroBlaze soft processor core. (2014). URL: http://www.xilinx.com/tools/microblaze.htm.