# Compiler-based Techniques to Secure Cryptographic Embedded Software against Side Channel Attacks

Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi *Member, IEEE*

***Abstract*—Side-channel attacks are a concrete and practical threat to the security of computing systems, ranging from high performance platforms to embedded devices. In this work, we will provide a brief systematization of the current existing approaches to analyze the side-channel vulnerability of an implementation, or automatically implement countermeasures, relying on methodologies typical of compiler systems. We will dedicate a spotlight to a significant progress in the countermeasures techniques which is represented by the application of dynamic compilation techniques to prevent a side-channel attacker from devising a model of the attacked application. We conclude the work highlighting promising research directions in this field.**

***Index Terms*—Side Channel Attacks, Automated Countermeasure Application, Code Morphing, Compilers**

## I. Introduction

THE ongoing process of deployment of a low-power computing and communication infrastructure denominated Internet of Things (IoT) calls for a systematic and effective approach at preventing security violations considering an attacker able to physically seize the target device. In such a scenario, a large variety of computing platforms, ranging from high-end embedded devices, such as home gateways and mobile phones, to extremely low-power microcontrollers, are required to perform tasks such as data encryption, endpoint authentication and digital signatures.

It is well understood that *side-channel attacks* (SCAs) represent one of the prime threats to the security of such systems due to both the low amount of resources required to lead them and the relative scarcity of systematically deployed countermeasures with respect to more common system-level attacks. A practical example of such a side-channel exploitation for an attack was given in [1], where the authors derive the encryption and authentication key of *smart bulbs* via differential power analysis, providing the practical grounds to allow them to deploy a worm on the said, ubiquitously present, embedded devices. The scientific literature proposing SCA countermeasures is rich, and provides effective solutions which are commonly tailored to a specific cryptosystem or platform, with the intent of maximizing their efficiency or minimizing their impact on computational performances and energy efficiency. However, the large variety of devices involved in the design space of IoT systems, paired with the significant amount of specific knowledge of the SCA framework highlight another requirement to progress towards widely deployed and efficient SCA countermeasures. Indeed, *automation* in side-channel analysis and countermeasure deployment is poised to be a key enabler in secure IoT development. In particular, automating the analysis and countermeasure deployment of software implementations of cryptographic primitives is an impactful research direction. Indeed, the higher turn-over of software versions with respect to hardware platforms, coupled with the reusable nature of software written in high level languages such as `C` or `C++` makes the automation of SCA countermeasure deployment appealing.

Arguably, the best established methodological infrastructure to automatically analyze and translate software implementations into executable objects is the one provided by classical compiler technologies. In this context, pioneering works tackling the execution-time side-channel such as [2], [3] have shown the effectiveness of employing methodologies from the compiler technology to both analyze and remove timing leakages from existing `C` code. An instance of such a systematic application is the removal of the archetypal execution time side-channel arising from unbalanced branches in an `if` construct, transforming the information leaking `if` into a predicated execution via arithmetic *if-conversion*s. Given the effectiveness of employing compiler based analyses and code transformations to automatically apply countermeasures, subsequent works [4] started applying local substitution techniques to assembly-level code representations, or relying on the type-system of a domain specific language translator [5] to automatically insert countermeasures against either power or Electro-Magnetic (EM) SCAs. This lead to a systematic use of the code transformation passes in a compiler to insert countermeasures, together with a dedicated dataflow analysis to optimize their insertion [6], [7]. A significant contribution from the use of dynamic compilation techniques to provide SCA countermeasures comes from the so-called *code morphing* approach. Introduced in [8], such a technique provides a countermeasure against power/EM SCAs by means of a periodical randomized recompilation of the cryptographic primitive implementation. The randomized recompilation modifies the executable code and effectively yields a moving target for the attacker, while preserving the code semantics.

**Contributions.** We provide a taxonomy of the current state-of-the-art of the application of compiler techniques to the analysis and automated securization of software cryptographic implementations. In particular, we split them in three main

G. Agosta, A. Barenghi, and G. Pelosi are with the Department of Electronics, Information and Bioengineering – DEIB, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133, Milano, Italy.
E-mail: *name.surname*@polimi.it

avenues, namely *side-channel vulnerability analyses*, *static code transformations inserting countermeasures*, and *code morphing*. The description of each avenue will provide an outlook on the existing techniques summarizing the core ideas, the employed methodologies, and reporting the degree of automation and effectiveness. We conclude our work pointing out challenges and opportunities for future research.

## II. Background

This section summarizes the background notions in compiler design, SCAs and SCA countermeasures needed to properly frame our systematization of the existing works.

**Compiler structure.** The traditional structure of a static compiler is the concatenation of three major components: the *front-end*, the *optimizer* and the *back-end*, which typically known as the compiler *pipeline*. The compiler propagates through the pipeline a representation of the analyzed source code of the program with a format known as the Intermediate Representation (IR). Multiple IRs can be employed along the compiler pipeline, depending on their suitability to perform a given analysis. The front-end tests the correctness of the source code according to the grammar describing the language, and usually builds a first IR of the input program to capture its syntactic structure in the so-called Abstract Syntax Tree (AST) [9]. Subsequently, the front-end runs on the AST a set of analyses such as type soundness checking and type inference, together with some simple code transformations (e.g., dead code elimination). The optimizer portion of the compiler pipeline receives the IR of the program from the front-end and performs a set of analyses and transformations aimed at improving the code performance or reducing the code size, while preserving its original semantics. It is typical to employ a Control-Flow Graph IR for this set of analyses, i.e., a representation of the program as a graph where each program statement corresponds to a node and the edges reflect data dependencies among them. The so-called *dataflow analysis* framework employs the Data-Flow Graph (DFG) representation of the program, i.e., a graph where each node is a program statement and the edges exiting from a node computing an intermediate variable are entering in the nodes where such a variable is required to be available to perform the computation. A dataflow analysis computes a property for each node of the DFG, e.g., whether or not a given definition of a variable (i.e., an assignment of the result of a computation to it) reaches a given point in the DFG without other definitions in between.

After analyses and code transformations are performed by the optimizer, the resulting IR of the program is passed to the compiler back-end. The compiler back-end translates the program IR into a sequence of instructions of the target Instruction Set Architecture (ISA). Such a process involves tasks such as *instruction selection* (picking a semantic preserving mapping between the IR statements and the ISA instructions), *register allocation* (allocating the variables to the ISA registers), and *instruction scheduling* to take advantage of the architectural features exposed by the target platform.

Dynamic compilation frameworks, often known as Just-In-Time (JIT) compilers follow conceptually the same pipeline of a static compiler, although they substantially differ in when the translation process is performed. Indeed, a static compiler executes the translation from the source language to the target ISA assembly all at once, thus performing no operations while the emitted program is run on the target platform. Dynamic compilers instead perform only a part of the compilation process, typically up to some point in the optimizer or the back-end and emit a machine-readable form of the IR known as *bytecode*. The compilation process is completed when the bytecode is run on the target platform. Dynamic compilers thus generate the binary code while running on the target platform, exploiting knowledge coming from profiling the execution of a previous version of generated binary code, improving execution speed and code size dynamically.

**Side Channel Attacks and Countermeasures.** A typical SCA workflow starts from modeling the side-channel behavior of a device assuming that either the plaintext or ciphertext of the implemented cryptosystem is available. The attacker chooses to model a small portion of the computation (e.g., $8$ bit in an `xor` operation) involving both the known value and the secret key. For each possible value of the key, a side-channel behavior model is derived and subsequently the actual side-channel (e.g., power consumption) is measured. All the key-dependent side-channel models are compared with the actual measured value, finding out which one is the best fit. Knowing the best fitting model reveals the actual value of the secret key. Since the side-channel measurements are affected by both random and systematic noise, the goodness of fit of a model to the actual device behavior is performed employing statistical tests, over a significant amount of side-channel samples.

Countermeasures aimed at protecting cipher implementations can be classified as *hiding*, *masking* and more recently *morphing*. Essentially, hiding [10] employs random delays or additional hardware energy dissipation elements to conceal the useful side-channel signal. Masking countermeasures [10]–[12] provide a formally proven lower bound on the computational effort required to the attacker to subvert the protection as they invalidate the correlation between the quantities employed to predict the power consumption and the actual values processed by the underlying device. In a masked implementation, each sensitive intermediate value is represented as split in a number of *shares*, $s$, which are all needed for its reconstruction. For example, $s$ shares are obtained as $s-1$ random values and the `xor` combination of them with the original input. The target algorithm is modified to perform the entire computation on the set of share-split values recombining them only at the end. The instantaneous power consumption is independent from the original (non-masked) value, as unpredictable random values are newly generated at each run of the cipher. The third category of SCA countermeasures, *morphing* was first proposed in [8] and relies on dynamically changing the code computing target cipher, while maintaining semantic equivalence. In the SCA context, a partial recompilation at run-time results in a moving target for the SCA attacker which can no longer determine a working side-channel model for the chosen portion of the computation as it may either be executed by means of a different sequence of instructions at each run. A typical way to evaluate the effectiveness of SCA countermeasures is to consider the amount of measurements to be collected required for an attacker to be successfully extracting the key. Indeed, provided a proper side-channel model is chosen (i.e., one taking into account the effects of the masking, if present), it is always possible to obtain a working SCA with an arbitrarily large amount of measurements. Indeed, it is not uncommon to evaluate the robustness of a SCA countermeasure in terms of the amount of Measurements To Disclose (MTD) the key. A system designer will thus choose a set of countermeasures which guarantee that the MTD will exceed the number of cryptographic primitive execution with the same key during the lifetime of the device.

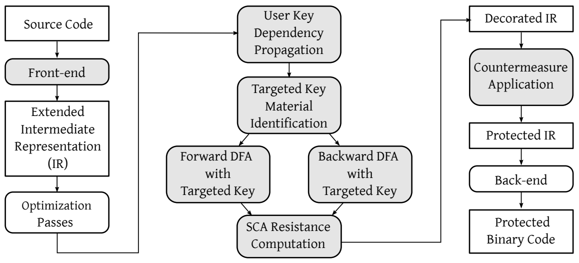
Source Code
Front-end
Extended Intermediate Representation (IR)
Optimization Passes
User Key Dependency Propagation
Targeted Key Material Identification
Forward DFA with Targeted Key
Backward DFA with Targeted Key
SCA Resistance Computation
Decorated IR
Countermeasure Application
Protected IR
Back-end
Protected Binary Code

Fig. 1. Representation of the LLVM compiler pipeline (added and modified passes in gray), including the passes implementing the security oriented dataflow analysis proposed in [6] and the additional passes where SCA countermeasures are automatically applied

## III. Compiler Techniques against SCAs

In this section we will provide an overview of the works which approached the problem of side-channel security in software employing compiler based approaches. In particular, we will classify the approaches in three main research directions: *side-channel vulnerability analyses*, *static code transformations inserting countermeasures*, and *code morphing*.

**Side Channel Vulnerability Analyses.** Typical, non-profiled SCAs apply a divide-et-impera strategy to reconstruct the cipher key through piecewise modeling of its effects on the side-channel behavior of the implementation. To this end, the attacker analyzes manually in detail the hardware/software stack on which the cryptographic primitive is implemented to determine an intermediate value for which a side-channel model can be computed guessing a small amount of key material. As a consequence, it is possible to provide a measure of the resistance against SCAs of a specific instruction considering the amount of key material involved in the computation of its result. Performing such an analysis by hand is both tedious and error prone as it is essentially replicating manually the computation of a *static dataflow analysis* performed by a compiler on a suitable representation of the cryptographic implementation (i.e., the *data flow graphs*).

The authors of [6] proposed a so-called *security oriented dataflow analysis* that quantifies the resistance against non profiled SCAs employing the amount of key material involved in the computation of a given intermediate value in the intermediate representation (IR) of the instructions of the cryptographic primitive at hand in the optimizer stage of the LLVM compiler framework. A summary of the modifications to the compiler pipeline done in [6] are reported in Figure 1, where gray boxes indicate modified or added passes. The proposed solution starts by analyzing a set of attributes employed to mark the plaintext and cipher-key in the source C code, providing the initial assignments for the *fixed point solver* of the dataflow analysis. In the same paper the authors also proposed to automatically insert a Boolean masking countermeasure only on the instructions that are computed to be below a given SCA resistance threshold. A dataflow analysis similar to the one of [6] was employed in [13] to compute an analogous score of resistance against active SCAs, in particular against the so-called *differential fault analysis* technique.

Static analysis techniques have been proposed also with the purpose of validating the soundness of a protected implementation. In particular, the *Quantified Masking Strength* (QMS) approach [14], [15] analyzes a masked implementation of a symmetric cipher translating the problem of quantifying the amount of randomness involved in each masked intermediate value in a formula solvable through a Satisfiability Modulo Theory (SMT) prover. The proposed method has the advantage of being exact in determining the strength of the applied protection, albeit at the cost of running an SMT on relatively large formulas. To the end of improving the running time of the QMS approach, the authors of [16] proposed to employ a specific *type-system* to augment C sources and perform automated deductions on whether an intermediate variable is uniformly distributed as the result of the application of a masking strategy or not. Such a static analysis technique is significantly faster than the QMS approach at the cost of precision in determining the protection status for some intermediate variables. Indeed, a hybrid approach resorting to a SMT solver approach only in the cases where the static analysis proposed by [16] is not accurate has the potential of being both accurate and reasonably fast. An approach similar to the one proposed in [14], targeted at detecting differential fault attack sensitive locations was proposed in [17], where the authors also automate the procedure to derive the differential relations required to perform a differential fault attack.

**Static Code Transformations to Insert Countermeasures.** The first instance of an automated countermeasure insertion against timing SCAs was reported in [2], where the authors devised a domain specific language for the description of asymmetric cryptographic primitives, emitting the code of an implementation automatically equipped to avoid secret-key dependencies of the computation time. A different approach, following an information theoretic analysis of the timing side channel leakage to provide an automated quantification of the said information was proposed in [18], [19]. A survey and critical analysis of the mentioned countermeasures and other compiler-based techniques to remove timing leakages is available in [3]. Considering power analysis based side-channels, applying a well defined masking countermeasure to a symmetric encryption algorithm implementation can be seen from a compiler standpoint as a *peephole* optimization, i.e., a local transformation on the code replacing an instruction sequence with a corresponding one, while preserving the end-to-end semantics of the transformed code. The locality of such a process is witnessed by [4], [20], where the authors operate a lexical substitution on assembly code to replace unprotected operations with protected ones, driven by a physical-measure derived leakage index. A different approach to automatically generate protected code is to employ a domain specific language and a dedicated translator as in [5], where a type inference mechanism was employed to determine which portion of the assembly emitted required masking. The first canonical compiler approach employing a dataflow driven code transformation applying a provably secure masking scheme was reported in [6]. The countermeasure application takes place in the compiler backend (see Figure 1), employing ad-hoc machine instructions to avoid countermeasure removal by the compiler backend. An integrated approach to the application of provably secure masking countermeasures considering a high level (i.e., AST level) representation of the source to be protected was proposed in [12]. In the aforementioned work, the authors defined a type-system and its inference rules to allow the automated computation of the sites where masked instructions should be inserted, as well as masks should be refreshed, to attain the so-called *strong non interference* property on the masked implementation. An alternative countermeasure strategy, relying on the insertion of dummy computations which result in fake SCA targets, was proposed in terms of an automatically performed code transformation pass in [21], [22]. The work exploits a dataflow representation of the algorithm to faithfully duplicate the behavior of the encryption acting with the actual secret key onto exact clones acting on a fake key acting as a red herring for the attacker.

Finally, considering the possibility of performing automated hardening of a software cipher implementation against fault attack, the authors of [23], tackle the issue of fault attacks aimed at altering the number of iterations executed or the exit condition in loops. Attacks of this kind can result in catastrophic security failures, e.g., obtaining a faulty AES ciphertext where the tenth round was skipped, together with the corresponding correct ciphertext allows to derive the entire secret key. The approach proposed in [23] adds redundant checks of invariants at the loop exit points (e.g., it adds redundant checks on the loop iteration counter) effectively hardening the code against single instruction skip faults. The authors of the paper also take care of preventing redundant code elimination passes from eliminating the duplicated checks, and avoiding the straightforward duplication of instructions with side effects. The resulting emitted code is validated against a simulated instruction skip fault model finding that 95% of the loops are effectively hardened, with 14% performance penalty.

**Code Morphing based Countermeasures.** The application of dynamic compilation techniques to software cryptographic primitives was proposed as a way to prevent an attacker from devising the side-channel behavior model required to perform an SCA. In [8] the authors proposed to embed a tailored dynamic re-compiler that changes the implementation of a block cipher at runtime, while preserving the semantic equivalence at the ends. The net effect of the dynamic recompilation is to significantly increase the MTD for the protected implementation, as a result of the high variability introduced in the actual computation. Whilst the ad-hoc dynamic compiler

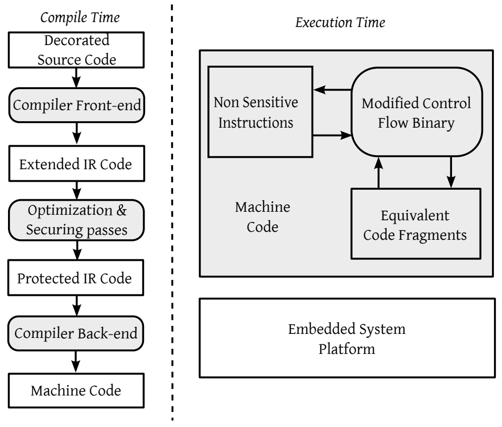

Fig. 2. High level overview of a code morphing countermeasure relying on code versioning [24]. The runtime code polymorphism is achieved through a randomized choice of code fragments to be executed at runtime among a set of statically generated ones

requires a significant amount of time to compute a new variant of the cipher implementation, it is possible to amortize this overhead over multiple encryptions, provided that the number of runs of the same variant of the cipher implementation is lower than the MTD by a safe margin. The authors of [8] report a $\approx 20\%$ performance overhead compared to an unprotected implementation, on their testbed platform.

With the intent of extending the approach proposed in [8] to platforms where the code segment cannot be written at runtime (e.g., microcontrollers) and reduce the overhead of the dynamic recompilation, the authors of [7], [24] proposed an alternative way to achieve runtime code variability, exploiting a code versioning strategy. The MEET approach, sketched in Figure 2, statically generates code variants for a given set of instructions, encased in a selection construct with a choice driven by a random number. The net effect at runtime is that, despite the code is entirely statically generated (hence removing the dynamic code generation overhead), the runtime execution path is still randomized and still providing a moving target to the side-channel attacker. In [7] refined and extended the versioning approach introducing also a lightweight mechanism to incrementally refresh values of masked look-up tables, showing that the MTD for the whole solution exceeds 100M.

The issue of reducing the overhead for dynamic code generation was tackled by the authors of [25], leveraging the dynamic compiler generation framework presented in [26]. The authors of [25] propose a solution, called Odo, where the dynamic compiler is generated as an instance tailored to the target cipher, effectively achieving faster code generation at runtime and small code size overheads. The same work leverages the spatial and temporal efficiency of the JIT compilers to enhance the morphed code by adding delay generation and insertion features, having a tunable random distribution, to the generated compiler. The random delays are generated as a randomized forward jump over a fixed length straight-line instruction sequence which does not act on the cipher state. The instructions composing the sequence are chosen in such a way that it is not possible to distinguish them from actual useful instructions present in the cipher. Finally, the presented solution also takes care of generating the instructions switching the memory protection mechanisms on platforms where a `write-xor-execute` policy is enforced on main memory, allowing the proposed solution to run also on write-protected code memory platforms.

## IV. Discussion and Future Directions

Notwithstanding the significant amount of research efforts and results in applying compiler techniques to analyze SCA vulnerabilities and automatically insert countermeasures, there are still a number of promising research avenues which deserve further attention. In the area of automated vulnerability analysis and countermeasure application, the current works either provide sound proofs of feasibility, with precise results, on small portions of symmetric cryptographic primitives (e.g., [14], [15]) employing a combination of tools, or provide approximate results on entire ciphers [6] completely integrated in a production grade compiler. Combining precision and acceptable speed into a unified framework, and implementing the corresponding solution into a widely accepted compiler toolchain is thus a natural development of the current state-of-the-art. Furthermore, the current analysis and countermeasure insertion techniques focus on an execution model which maps one-to-one the architecture level view onto the actual computation being performed. Such a model was proven to be potentially mismatched with the actual execution and side-channel leakage model in case of superscalar architectures [27]. An interesting research direction in this respect is to integrate the microarchitectural information on the compilation target into the side-channel analysis and countermeasure insertion passes in the compiler, matching what is currently a well established practice to obtain high performance compiled code.

Considering the case of code morphing countermeasures, a challenging and interesting open question is to provide formal security guarantees on the effectiveness of code-polymorphism based countermeasures. Indeed, while [8] still stands unbroken, and the table based method [24] passed a non-specific $t$-test run with 100M samples, no formal framework describing the lowering of the signal-to-noise ratio provided by morphing countermeasures has been developed. From a performance standpoint, the overhead imposed by dynamic recompilation or code versioning, although acceptable [7], [24], [25], is still non negligible. Investigating more efficient techniques to perform code morphing without sacrificing the provided security margin is still an interesting topic for research. In particular, a possible research direction is the one of adapting the traditional strategies for bytecode execution acceleration to perform code morphing.

## References

[1] E. Ronen, A. Shamir, A. Weingarten, and C. O'Flynn, "IoT Goes Nuclear: Creating a Zigbee Chain Reaction," *IEEE Security & Privacy*, vol. 16, no. 1, 2018.

[2] M. Barbosa, A. Moss, and D. Page, "Constructive and Destructive Use of Compilers in Elliptic Curve Cryptography," *J. Cryptology*, vol. 22, no. 2, 2009.

[3] J. V. Cleemput, B. Coppens, and B. D. Sutter, "Compiler mitigations for time attacks on modern x86 processors," *TACO*, vol. 8, no. 4, 2012.

[4] A. G. Bayrak, F. Regazzoni, P. Brisk, F. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proc. of DAC 2011*, 2011.

[5] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler Assisted Masking," in *CHES 2012*, ser. LNCS, vol. 7428. Springer, 2012.

[6] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, "Compiler-based side channel vulnerability analysis and optimized countermeasures application," in *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, 2013.

[7] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 34, no. 8, 2015.

[8] G. Agosta, A. Barenghi, and G. Pelosi, "A Code Morphing Methodology to Automate Power Analysis Countermeasures," in *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*. ACM, 2012.

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[10] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[11] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, "Consolidating masking schemes," in *CRYPTO 2015*. Springer, 2015.

[12] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, P. Strub, and R. Zucchini, "Strong Non-Interference and Type-Directed Higher-Order Masking," in *Proc. of CCS 2016*. ACM, 2016.

[13] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "Differential Fault Analysis for Block Ciphers: an Automated Conservative Analysis," in *Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, UK, Sep. 9-11, 2014*. ACM, 2014.

[14] H. Eldib, C. Wang, M. Taha, and P. Schaumont, "QMS: Evaluating the side-channel resistance of masked software from source code," in *Proc. of the 51st Annual Design Automation Conference*. ACM, 2014.

[15] H. Eldib, C. Wang, M. M. I. Taha, and P. Schaumont, "Quantitative Masking Strength: Quantifying the Power Side-Channel Resistance of Software Code," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 34, no. 10, 2015.

[16] J. Zhang, P. Gao, F. Song, and C. Wang, "SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks," in *30th International Conference on Computer Aided Verification (CAV 2018)*, ser. LNCS, vol. 10982. Springer, 2018.

[17] J. Breier, X. Hou, and Y. Liu, "Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, 2018.

[18] B. Köpf and D. A. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proc. of CCS 2007*. ACM, 2007, pp. 286–296.

[19] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 4:1–4:32, 2015.

[20] A. G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F. Standaert, and P. Ienne, "Automatic Application of Power Analysis Countermeasures," *IEEE Trans. Computers*, vol. 64, no. 2, 2015.

[21] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "Information leakage chaff: feeding red herrings to side channel attackers," in *Proceedings of the 52nd Annual Design Automation Conference, DAC'15, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015.

[22] ——, "Reactive side-channel countermeasures: Applicability and quantitative security evaluation," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 62, 2018.

[23] J. Proy, K. Heydemann, A. Berzati, and A. Cohen, "Compiler-Assisted Loop Hardening Against Fault Attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, 2017.

[24] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "A Multiple Equivalent Execution Trace Approach to Secure Cryptographic Embedded Software," in *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*. ACM, 2014.

[25] N. Belleville, D. Couroussé, K. Heydemann, and H.-P. Charles, "Automated Software Protection for the Masses Against Side-Channel Attacks," *ACM Trans. on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, 2018.

[26] H. Charles, D. Couroussé, V. Lomüller, F. A. Endo, and R. Gauguey, "deGoal a Tool to Embed Dynamic Code Generators into Applications," in *CC 2014.*, ser. LNCS, vol. 8409. Springer, 2014.

[27] A. Barenghi and G. Pelosi, "Side-channel security of superscalar CPUs: evaluating the impact of micro-architectural features," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 2018.