Automated Masking of Software Implementations on Industrial Microcontrollers

Arnold Abromeit*, Florian Bache†, Leon A. Becker†, Marc Gourjon‡§, Tim Güneysu†¶
Sabrina Jorn*, Amir Moradi†, Maximilian Orlt||, Falk Schellenberg**
*TÜV Informationstechnik GmbH, Essen, Germany
†Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
‡Hamburg University of Technology, Hamburg, Germany
§NXP Semiconductors Germany GmbH, Hamburg, Germany
¶Cyber Physical Systems, DFKI GmbH, Bremen, Germany
||Technische Universität Darmstadt, Germany
**MPI-SP, Germany
Email: {florian.bache, tim.gueneysu}@rub.de

Physical side-channel attacks threaten the security of exposed embedded devices, such as microcontrollers. Dedicated countermeasures, like masking, are necessary to prevent these powerful attacks. However, a gap between well-studied leakage models and observed leakage on real devices makes the application of these countermeasures non-trivial. This work provides a gadget-based concept to automated masking covering practically relevant leakage models to achieve security on real-world devices. We realize this concept with a fully automated compiler that transforms unprotected microcontroller-implementations of cryptographic primitives into masked executables, capable of being executed on the target device.

In a case study, we apply our approach to a bitsliced LED implementation and perform a TVLA-based security evaluation of its core component: the PRESENT s-box.

I. Introduction

Independent of a cryptographic scheme's mathematical security, side-channel attacks exploit unintentional information channels to potentially extract the involved secret key, e.g., by measuring the power consumption of a device-under-test. Its introduction in 1999 by Kocher [1] triggered flourishing research in sophisticated attacks and developing corresponding countermeasures. Yet, implementing countermeasures such as masking is still a tedious and error-prone task, especially for software countermeasures running on embedded devices. This is mainly due to the following characteristics:

The practical security of a countermeasure cannot be evaluated at a high abstraction level, such as C-code. This is due to the various translation steps to machine code that might add, remove, or reorder instructions along the way. Indeed, there are numerous examples of compilers breaking otherwise secure masking schemes [2], [3].

Further, the device-under-test's internal architecture adds additional *device-specific leakage* way beyond a simple Hamming-weight or value-based leakage model.

For example, a register update will produce the Hamming-distance between the new and the old value. Correct first-order masking becomes insecure in practice if two shares k_0, k_1 of a secret value k are combined with such leakage behavior, and information on the secret k is leaked. Indeed, leakage across instructions invalidates not only the theoretical assumption of "independent leakage" but also prevents the re-use of security unaware compiler back-ends [3]–[6]. Most notable, this device-specific behavior also applies for different implementations of the same architecture (e.g., RISC-V), leading to broad diversity of physical side-channel behavior among devices even when executing the very same code.

A. Contribution

We tackle these issues by introducing a dependable compiler that automatically and effectively protects insecure code. The compiler provides automated side-channel resilience in practice by combining careful device-specific masking with secure code transformations. Our code transformations are independent of device-specific leakage behavior and allow users to adapt the approach to new devices. Informally, the approach consists of replacing vulnerable machine-code with "gadgets", which provide the same functionality in a power side-channel resilient manner. We formally verify our gadgets' security in precise device-specific leakage behavior models and describe the modeling process.

A proof-of-concept implementation of our approach for Arm Cortex M0+ microcontrollers is developed and applied to a bitsliced implementation of the LED block cipher. Finally, we demonstrate the effectiveness of our approach by a practical security evaluation of physical measurements.

B. Related work

Many papers try to automate certain aspects of the implementation of countermeasures. We focus on software countermeasures and briefly discuss the shortcomings of existing work in the following.

a) Insufficient leakage assumption for practical resilience: Moss *et al.* provided one of the first compilers for unprotected C-code to masked assembly [7] but only rely on first-order masking and do not consider device-specific leakage. Eldib *et.* Wang [8] use synthesis to mask C-Code in LLVM Intermediate

978-3-9819263-5-4/DATE21/©2021 EDAA

Representation (IR). Most notable, they indeed report that compiler transformations break the correctness of masking.

b) Dependence on security of off-the-shelf compilers: Barthe *et al.* develop a fast compiler masking C-Code at higher-order in [9]. They depend on an off-the-shelf compiler to produce executable code. This likely breaks security as reported in [2], [3]. Further, they do not consider device-specific leakage. Likewise, Agosta *et al.* [10] employ precise information flow analysis within LLVM but suffer from the same shortcomings, i.e., a potential break at LLVM's back-end and neglected device-specific leakage. Recently, Belleville *et al.* automate application of first-order masking, capable of masking tables [2]. It is implemented as middle-end compiler pass on LLVM IR and uses formal verification on machine code to analyze which back-end(s) potentially break the countermeasure. Indeed, they report at least one harmful back-end pass for ARM architectures. However, their analysis is restricted to a basic value leakage model under the "independent leakage assumption" and is not backed by physical security assessment. However, this is crucial as multiple passes such as instruction scheduling and register allocation potentially break security in presence of leakage spanning across instructions. Instead, our approach consists of assembly transformation to prevent flaws from regular compilation.

Wang *et al.* [11] repair register allocation of LLVM to prevent leakage arising from register re-use, but assume that the input code is already masked. Further, the modification is specific to the leakage model and cannot be easily extended to cover leakage across instructions. We note that fixing compiler passes according to model assumption most likely renders the entire compiler device-specific. Instead, our compiler passes are independent of any leakage model assumptions.

Bayrak *et al.* automatically insert first-order Boolean masking in machine-code [12]. While they do not suffer from potentially harmful compiler-backends, device-specific leakage is again out-of-scope. They also discuss pairing physical evaluation with compilation to focus solely on detectable leakage. Unfortunately, this mandates the presence of a measurement setup and respective expertise during compilation which somewhat contradicts the purpose of automated security.

Conceptually, practical resilience can also be achieved by combining automated masking with a subsequent automatic repair: Potentially insecure output of the first step could be secured by inserting more countermeasures, as e.g. discussed in [13]. One inherent issue here is that a repair based on inserting additional instruction is not always possible, especially if shares are combined during computation by compiler optimizations. Instead, our work provides systematic protection resulting in practically-resilient implementations while only relying on device-specific gadgets.

II. CONCEPT

Our core concept consists of replacing vulnerable machine-code computing on sensitive data by invocations of gadgets that compute securely on masked data. In the end, a protected executable is returned, which can directly be run on the targeted platform. The overall compilation, depicted in Fig.. 1, consists of three stages: (1) compilation of annotated but unprotected code with an optimizing off-the-shelf C-compiler (gcc), (2) information and control flow analysis (IFA/CFA) to mark sensitive computations for transformation, and (3) the actual substitution of insecure instructions by secure gadgets.

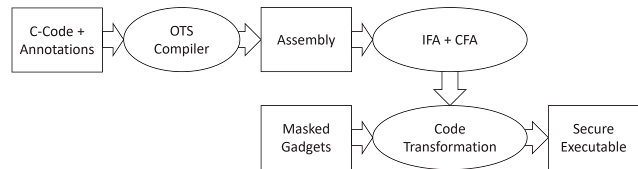

Fig. 1. High-level structure of our proposed automated masking approach.

The annotation for stage (1) is very simple and only consists of a specification of the sensitive inputs to the algorithm, e.g., the plaintext and key.

Stage (3) makes use of a library of gadgets specifically designed to achieve device-specific side-channel resilience. Pointers replace sensitive data in registers with masked data stored in memory. The compiler inserts code to handle memory, data, masking, and demasking of sensitive data, randomness, and the invocation of secure gadgets so that the same functionality is computed in a side-channel resilient manner.

The gadget-based approach succeeds whenever the sensitive part of the source program can be expressed by operations for which gadgets exist. We focus on bitsliced ciphers, which can be represented using Boolean masking. However, the concept could also secure other types of the input program, such as add–rotate–xor (ARX) ciphers when the required gadgets are provided (e.g., for addition).

The realization of such compilation results in two tasks: (1) construction of device-specific gadgets (Section IV) (2) vulnerability analysis and code rewriting (Section V). Task (1) is performed by security experts prior to compilation and is re-used among code to be compiled. Task (2) results in a fully automated compiler using the gadgets of Task (1).

III. SECURITY IN PRACTICE

Our goal is to reliably establish baseline protection against simple attacks such as Simple Power Analysis (SPA), Correlation Power Analysis (CPA), and first-order Differential Power Analysis (DPA) [1]. Physical side-channel attacks can be classified according to the number of sample points they exploit per measurement of execution and the amount of measured executions in total. In the following, we establish resilience against attacks exploiting a single attacker chosen sample point for large amounts of repeated executions. Such protection forces adversaries to launch more complex attacks that require expertise to exploit multiple sample points. We mandate resilience to be established and evaluated without device-dependent noise or additional countermeasures (e.g., shuffling), which perturb physical measurements but can be removed by pre-processing the measurements.

First-order Boolean masking can provide protection in this setting as sensitive data k is split into shares k_0, k_1 such that $k_0 \oplus k_1 = k$ where one share is a uniform random value.

Information on both shares must be recovered by an adversary to recover the masked secret k due to the involved randomness. Attacks exploiting a single measurement point cannot be successful if the two shares remain separated during computation and no measurement sample reveals information on more than one share [14]. The latter constraint is important for resilience in practice as processors are well-known to emit leakage, combining shares during the execution of software [3]–[6]. In contrast to existing work, we take additional effort to ensure both constraints are met after compilation.

In [15] we show a link between side-channel resilience in practice and "Threshold Non-Interference" (NI), a formal notion of side-channel security, in detailed models of observable side-channel information (denoted "leakage models"). First order NI ensures that each observable side-channel information (leakage) depends on at most one share of each masked secret. The formal notion of security implies our intended degree of practical security in case the leakage model contains all information that can be gained by one physical measurement sample. Slightly different, we exploit the indicated link to prevent human errors and speed up development: we consolidate experienced side-channel behavior in leakage models and use the formal verification tool scVerif [15] to ensure the absence of vulnerabilities arising from *known* leakage.

We briefly explain how our models are constructed, which consists of two tasks (a) deciding whether modeled leakage is present on a device and (b) systematizing unknown leakage behavior into formal models (the formal model is presented in [15]).
Validating the presence of modeled leakage can be done by constructing implementations that emit detectable leakage in physical measurements (e.g., using TVLA, Section VI-A). The leakage behavior under test must be activated while all other known leakage behavior must not cause detectable leakage. This can be achieved by filling the device state with secret-independent (e.g., constant) data, except for two carefully placed shares, which are combined by the leakage behavior under test (see [4] for a detailed discussion). The tool scVerif can be used to verify that known leakage behavior is not causing vulnerabilities in such programs. False positives are possible when unknown leakage behavior is triggered. This can be detected once the model becomes more complete as inconsistencies arise during the validation of another leakage (which is why model construction is an iterative process).

Systematization of unknown leakage behavior is not straightforward: Whenever leakage is detected, informally secure implementations, a hypothetical leakage is modeled according to human understanding. This can be supported by verification as the newly modeled leakage should lead to verification failure in the implementation. Often it is sufficient to revise existing leakage behavior. Subsequently, the validation strategy (a) is employed for the hypothetical leakage to validate its presence.

Our starting point was to validate the presence of published leakage behavior (e.g. [3]–[6]), which reduced the likeliness of false positives and quickly led to an almost robust model. We validated the completeness of our models by constructing masked implementations which are provably secure in our model and perform a physical assessment to detect gaps.

IV. Gadget Design

The gadgets must provide the functionality of instructions supported by a microcontroller while being side-channel resilient in practice. The gadgets are handed over to the compiler in the form of a library, which allows it to construct them for a specific microcontroller and mitigate device-specific leakage behavior (known as "hardening").

We briefly explain the process of hardening: Device specific leakage behavior allows observing information on more than one share within a single measurement point. Hardening means to prevent such vulnerabilities by breaking the leakage into separate measurement points by either (a) reordering instructions or operands and (b) inserting secret independent instructions (e.g., `NOP`). In addition, the gadgets must ensure that no sensitive data is left as residue, neither in memory nor in registers.

As the compiler is operating in a restricted setting, some technical subtleties arise: (1) The compiler operates on pre-sampled randomness provided in the form of a pointer, which must be correctly maintained by the gadgets to prevent vulnerabilities from randomness re-use. (2) The gadgets operate on dedicated pointers to memory for input and output, and in-place modification in the form of equal input and output pointer is possible, which requires loading shares before writing outputs. (3) For some leakage behavior, it is required to access secret independent memory, which cannot be provided by the leakage independent compiler passes.

An inherent difficulty is the execution of gadgets in contexts which are determined only after construction is complete (i.e., during compilation) as it corresponds to dynamic composition of masked implementations: Vulnerabilities might arise when a set of shares is reused as input to multiple gadgets as most composition strategies for masked gadgets assume independently shared inputs (e.g. t-SNI [9]). The easiest approach is to refresh all inputs within each gadget. A more efficient approach is to analyze which shares are reused during compilation and only selectively refresh these shares. Another problem arises when gadgets receive two inputs pointing to the same shares. This case can happen in non-optimized code when protecting `AND R0 R0` the corresponding gadget potentially produces vulnerable leakage. Again, this is easy to circumvent within the compiler or by refreshing inputs internally. Our compiler refreshes all shared values used more than once.

For our case study, the compiler requires gadgets to replace the assembly instructions `EORS` (exclusive or), `ANDS` (logical conjunction), `MOV` (copy), `MVNS` (copy negation), `BICS` ($a \& \neg b$), as well as gadgets to copy and refresh masked values which are adapted from [15].

V. Code Transformation

In this section, we describe the overall process that produces a masked executable from an unmasked C-code. The input program needs to be annotated in order to enable the tainting phase of the code transformation. This annotation

consists of one or more in-line assembly statements marking the initial sensitive values in the interface functions of the code (e.g. `main` or `encrypt`). In the first step, an off-the-shelf C-compiler for the target processor is used to generate optimized assembly code from the input C-program. Specifically, the bare-metal ARM-GCC compiler is invoked with the command `arm-none-eabi-gcc {source file} -S -mthumb -mcpu=cortex-m0plus -O3`. The operations described in the following sections are then performed on the assembly code.

A. Parsing and Tainting

In the parsing step of the transformation, we extract the hierarchical structure and control flow of the input assembly program while keeping all low-level information including opcodes and register allocation used in every instruction. Our implementation relies on several assumptions to implement this: (1) The input program is composed of data and a text segment and includes a prologue and an epilogue consisting of directives. (2) The data segment occurs once and does not intersect with the text segment. (3) The text segment is divided into one or more functions that do not intersect. Each section is marked by a prologue and epilogue of directives. (4) Every function has exactly one return instruction. (5) The number of loop iterations must be known at compile time. (6) Access to memory locations which depends on sensitive data is not supported. While assumptions 1 to 3 are enforced by the compiler 4 to 6 must be guaranteed by the input program. Note that, while assumption five seems strong, the targeted algorithms, such as block ciphers, do generally not need variable loops.

On the highest level, the data and code segments of the program are identified and then processed separately. The data segment is then scanned for data objects and their meta-information is stored. In the next step, basic blocks followed by functions are identified in the text segment. To this end, the assembly code is parsed line by line, the instruction class is determined (e.g. load/store, arithmetic, branch) and the control flow graph is constructed iteratively. Additionally, conditional blocks and program loops are identified in this process.

The following tainting step is necessary to identify the instructions that operate on sensitive data as well as sensitive data itself. To this end, our compiler creates a tainting context that holds the complete program state including registers and the relevant memory segments. The context is then copied for each encountered basic block to prevent side effects caused by splits in the program flow. Note that the tainting process is also able to determine the size of arrays that contain sensitive data by relying on assumption (5) from above.

B. Instruction Replacement

In the final step of the code transformation process, the previously tainted instructions are replaced by secure gadgets. To this end, every sensitive value is replaced by a pointer to the first share of its masked realization. The required memory for this process is allocated on the stack or a data segment, depending on context. There is no need to store the size of a value or additional pointers to subsequent shares of the same variable, as the transformation assumes that all variables have a size of one word. Therefore, the other shares of the value can be placed successively in memory as depicted in Fig. 2. For

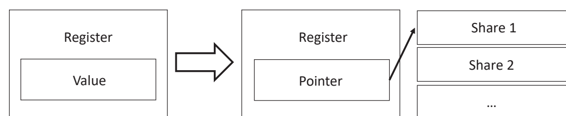

Fig. 2. Masking of sensitive variable in register.

every array in the input data segment that was tainted a new data object is allocated to store the shared data. Subsequently, the original array is replaced by pointers to the shares. The compiler also allocates a memory segment used to hold an entropy pool and a global pointer to the first entropy word. The gadgets use this pointer at runtime to obtain entropy and increment it after each access. Note that the compiler does not create code that generates the entropy needed by the secure gadgets, giving the choice about the source of randomness (e.g., internal source or external input) to the user. The final global transformation inserts code to mask and unmask the sensitive data at top-level functions.

The transformation of the program code is then applied on function level, basic-block level and finally instruction level.

a) Function Level: The following transformations are applied to the functions: (1) Reservation of additional stack space for temporarily storing shares of sensitive variables. In our proof-of-concept implementation, sufficient space for shared values corresponding to all general purpose registers is allocated. The purpose of this ”shadow register” for masked values will be explained in Sect. V-B0c. In a future optimization, the required stack space could be reduced by analyzing which registers do specifically need this inside each function. (2) A stack base-pointer is created which allows referencing the temporary storage described above. (3) The link register is pushed. This is necessary because the gadgets are called using linked branches and will therefore overwrite the original link register. (4) Additional push and pop instructions are inserted in order to save registers that are used by the transformation tool. (5) Some minor optimizations are performed , e.g., the removal of successive `mov` instructions with the same source and target.

b) Basic Block Level: The transformations on the basic block level counteract the code expansion produced by the masking compiler. This is necessary as the range of simple conditional branches and unconditional branches on the target platform is only $254\,\mathrm{B}$ and $2\,\mathrm{kB}$ respectively. The code expansion can prohibit the use of either type of branch instruction. Therefore, the transformation replaces branches with linked long jumps that have a range of $16\,\mathrm{MB}$, except if the branch targets its own basic block and the block is sufficiently small.

c) Instruction Level: In the final transformation step, the actual instruction replacement is performed. As the gadgets expect their parameters and store the result in specific registers,

blending code is inserted before and after the gadget call which moves the data to the correct registers. As this blending code plays an important role in overhead generation (both in time and space) a future optimization could combine ”glue”-code of consecutive gadget calls. If the vulnerable instruction is not a `mov`, `load` or `store` instruction it is replaced by a call to its corresponding secure gadget at this point. For the remaining instructions, the shadow register file must be used. This is necessary because an unmasked value in the input program corresponds to a pointer in the masked program as explained above. If `load`, `store` or `mov` instruction directly operate on these pointers, semantic equivalence of the input and output program can not be assured. To see this, consider the following sequence of instructions operating directly on unmasked values: mov r1, r0; xor r1, r1, r2.

Here, the value of `r0` should not change. However, if the values are replaced by pointers and the `xor`-operation is replaced by a gadget operating on the referenced data, `r1` would now point to the (masked) value `r1 xor r2`. To prevent this, the move instruction is modified in the following way: The shared data pointed to by `r0` is copied to the shadow register file on the stack corresponding to `r1` and a pointer to the first share is stored in `r1`. Loading and storing data produces similar problems, that are solved with the same approach.

C. Security of Transformations

The compiler produces secure implementations when provided with secure gadgets. Two cases have to be considered; leakage arising during execution of gadgets and in compiler generated code. The former is true by definition. The latter holds since the compiler generated code operates on pointers to sensitive data but never accesses the data itself without using secure gadgets (including freeing of memory). The argument relies on the fact that no hidden memory accesses are performed based on pointer operations. Such behavior would become visible in the physical evaluation of securely masked gadgets and thus poses no immediate threat.

VI. Case Study

We chose a bitsliced implementation of the LED-128 block cipher [16] as our case study target. LED-128 is based on a Substitution-Permutation Network and uses a 128 bit key to encrypt a 64 bit plaintext to the ciphertext in $12 \cdot 4$ rounds. Each round consists of an addition of round constants, the non-linear `SubCells` layer relying on the 4-bit PRESENT s-boxes, followed by the `ShiftRow` and `MixColumnsSerial` layers. After every four rounds, a round key is added to the state. Our source C-program realizes LED as a bitsliced implementation using the whole 32-bit register width of the target platform.

A. Security Evaluation

We perform a physical validation of the s-box implementation masked by our compiler, analyzing the effectiveness of the masking against both power and electromagnetic (EM) side-channel attacks.

The evaluation is performed on an industrial Arm Cortex M0+ Microcontroller (MCU), specifically an FRDM-KL82Z prototyping board with removed capacitors. The supply voltage is set to $1.8\,\mathrm{V}$ and clock frequency of $96\,\mathrm{MHz}$ is used.

For measuring, a Teledyne LeCroy HDO6154 oscilloscope with a sampling rate of $2.5\,\mathrm{GSamples/s}$ is used. The power consumption is measured with a Teledyne LeCroy AP033 active differential probe placed in the MCU's power supply line (bandwidth $500\,\mathrm{MHz}$). EM emanation is measured with Langer EMV ICR HH500-75 IC near field probe (coil diameter $500\,\mu\mathrm{m}$, bandwidth $1\,\mathrm{GHz}$).

A lateral scan over the front side of the MCU is performed to find the optimal position for the EM probe. The resulting heatmap reflects whether the EM signals acquired during the computations show clear, high peaks or whether no structures related to the computation can be observed. Step size for the lateral scan is set to one-tenth of the EM probe's coil diameter to have sufficient lateral resolution. In subsequent , the EM probe is positioned over the location providing the optimal EM signal.

To identify the period of s-box processing in the measured traces, spill actions just before and after the s-box calculations are introduced. These serve as triggers near the start of the s-box calculation by intentionally producing high values in the following t-tests (for test calibration) and to characterize the signal-to-noise ratio. Despite accurate triggers, power and EM traces are synchronized based on reference patterns and determine the highest cross-correlation of optimally shifted traces against this reference pattern.

Schneider *et al.* provide a detailed discussion of the underlying analysis method, capable of robustly assessing the physical vulnerability of cryptographic devices. Two-tailed t-tests are performed on sets of power consumption and EM emanation traces separately. For this, one million measurements are taken with alternating inputs, i.e., either the input was chosen randomly, or a fixed input was sent to the s-box calculation (known as fixed vs. random t-test). The critical t-value limits are shown by dashed lines in the results in Figure 3 and Figure 5. T-values above or below these lines indicate significant leakage, i.e., the detectable difference between the two groups respective their distributions and therefore distinguishability of calculations using fixed inputs and calculations using random inputs. The method allows detecting leakage without making assumption on a specific leakage model, yielding a robust indication that the implemented countermeasure efficiently prevents leakage.

The analysis is performed by sending fixed input bytes (respective random input bytes) together with required entropy bytes to the protected s-box implementation on the MCU. No leakage should be visible if the implementation is properly masked.

The unprotected implementation shows detectable leakage after just 500 measurements as depicted in the results of the t-test in Figure 3.

For the s-box implementation masked by our compiler, one

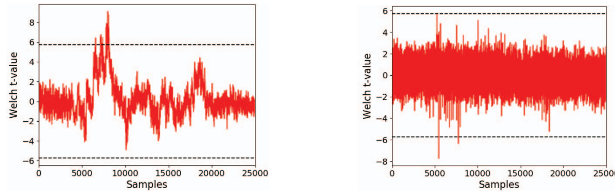

Fig. 3. t-test results for power and EM traces in unprotected s-box

million traces were recorded, and some sample traces are depicted in Figure 4. The results of leakage detection visible in Figure 5 show that there is only at the beginning and end of the test program an indication of exploitable leakage, due to the artificially introduced spill actions. However, there is no indication of any relevant leakage during the masked s-box computation despite the high number of measurements.

We conclude that without masking, there is extreme data input dependence visible after only 500 traces. After automated protection with our tool, no data dependency is visible in both the power and EM domains, even when using 10^6 traces.

B. Performance

In this section we provide the relevant overhead produced by the proof-of-concept implementation of our proposed approach to automated masking. When applied to our LED benchmark implementation, our automated masking tool identifies $57.4\,\%$, i.e., 666 of the 1161 instructions in the unmasked input program's assembly code as sensitive. After masking, the number of instructions in the code is increased to 8418 (x7.3) and the runtime by x39.3. When our tool is applied to just the s-box, 63 out of 73 ($86.3\,\%$) instructions are sensitive. In this case, the number of instructions is increased to 835 (x11.4) and the runtime by 11.7. Note that the timing overhead includes the time needed for masking and unmasking the input data. The required randomness is and $599.6\,$kB and $324\,$B respectively.

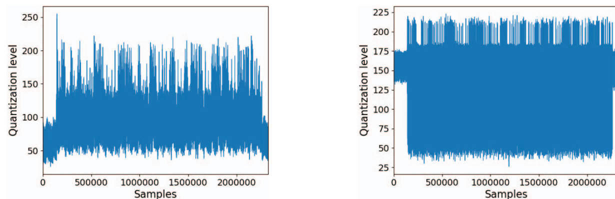

Fig. 4. Exemplary power (left) and EM trace (right) of the protected s-box

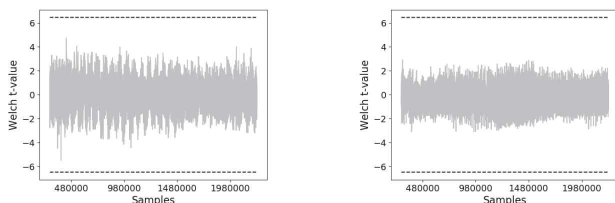

Fig. 5. t-test results for power and EM traces in the protected s-box

VII. Conclusion

We present a dependable compiler providing automated protection against first-order power side-channel attacks. The compiler substitutes vulnerable machine-code with secure gadgets specifically designed to mitigate vulnerabilities arising from device-specific leakage. Further, the security of our compiler cannot be invalidated by optimizing compilation passes which might break otherwise secure masking. The devised transformations are independent of specific side-channel behavior and thus can to be ported to devices with more diverse side-channel behavior. Our gadgets are formally verified in precise models, and the resilience in practice is successfully evaluated in physical measurements.

Since the approach naturally elaborate to higher-orders of security, we would like to elaborate on it in future work.

Acknowledgments

This work was supported in part by the German Research Foundation under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, the Emmy Noether Program FA 1320/1-1 and the Federal Ministry of Education and Research of Germany through the VeriSec Project (16KIS0603, 16KIS0601K, 16KIS0602, 16KIS0604, and 16KIS0634).

References

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO*, ser. LNCS. Springer, 1999.

[2] N. Belleville *et al.*, "Maskara: Compilation of a Masking Countermeasure with Optimised Polynomial Interpolation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[3] J. Balasch *et al.*, "On the Cost of Lazy Engineering for Masked Software Implementations," in *CARDIS*, ser. LNCS. Springer, 2014.

[4] K. Papagiannopoulos and N. Veshchikov, "Mind the Gap: Towards Secure 1st-Order Masking in Software," in *COSADE*, ser. LNCS. Springer, 2017.

[5] D. McCann, E. Oswald, and C. Whitnall, "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages," in *USENIX*, 2017.

[6] Y. L. Corre, J. Großschädl, and D. Dinu, "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors," in *COSADE*, ser. LNCS. Springer, 2018.

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

[8] H. Eldib and C. Wang, "Synthesis of Masking Countermeasures against Side Channel Attacks," in *CAV*, ser. LNCS. Springer, 2014.

[9] G. Barthe *et al.*, "Strong Non-Interference and Type-Directed Higher-Order Masking," in *ACM CCS*. ACM, 2016.

[10] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, "Compiler-based side channel vulnerability analysis and optimized countermeasures application," in *DAC*. ACM, 2013.

[11] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ESEC/SIGSOFT FSE*. ACM, 2019.

[12] A. G. Bayrak *et al.*, "Automatic Application of Power Analysis Countermeasures," *IEEE Trans. Computers*, 2015.

[13] M. A. Shelton *et al.*, "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers," *CoRR*, 2019.

[14] Y. Ishai, A. Sahai, and D. A. Wagner, "Private Circuits: Securing Hardware against Probing Attacks," in *CRYPTO*, ser. LNCS. Springer, 2003.

[15] G. Barthe *et al.*, "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification," *IACR Cryptol. ePrint Arch.*, 2020.

[16] J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw, "The LED Block Cipher," in *CHES*, ser. LNCS. Springer, 2011.