RiCaSi: Rigorous Cache Side Channel Mitigation via Selective Circuit Compilation

Heiko Mantel, Lukas Scheidel, Thomas Schneider, Alexandra Weber(✉), Christian Weinert, and Tim Weißmantel

Technical University of Darmstadt, Darmstadt, Germany
{mantel,weber,weissmantel}@mais.informatik.tu-darmstadt.de,
{scheidel,schneider,weinert}@encrypto.cs.tu-darmstadt.de

Abstract. Cache side channels constitute a persistent threat to crypto implementations. In particular, block ciphers are prone to attacks when implemented with a simple lookup-table approach. Implementing crypto as software evaluations of circuits avoids this threat but is very costly.

We propose an approach that combines program analysis and circuit compilation to support the selective hardening of regular C implementations against cache side channels. We implement this approach in our toolchain RiCaSi. RiCaSi avoids unnecessary complexity and overhead if it can derive sufficiently strong security guarantees for the original implementation. If necessary, RiCaSi produces a circuit-based, hardened implementation. For this, it leverages established circuit-compilation technology from the area of secure computation. A final program analysis step ensures that the hardening is, indeed, effective.

1 Introduction

Cache side channels are unintended communication channels of programs. Cache-side-channel leakage might occur if a program accesses memory addresses that depend on secret information like cryptographic keys. When these secret-dependent memory addresses are loaded into a shared cache, an attacker might deduce the secret information based on observing the cache.

Such cache side channels are particularly dangerous for implementations of block ciphers, as shown, e.g., by attacks on implementations of DES [58,67], AES [2,11,57], and Camellia [59,67,73]. A key reason why block-cipher implementations are vulnerable to cache-side-channel attacks is that they traditionally use secret-dependent accesses to lookup tables in memory. For instance, the original AES specification [20] recommends lookup tables to increase performance. Such lookup-table-based AES implementations are still available in many crypto libraries, including, e.g., OpenSSL [55] and mbedTLS [7].

To avoid cache-side-channel leakage, block ciphers can be implemented as circuits that are evaluated in software, e.g., using the bitslicing technique [12,35,47]. For instance, Matsui and Nakajima [47], as well as Käsper and

© Springer Nature Switzerland AG 2020
S. Krenn et al. (Eds.): CANS 2020, LNCS 12579, pp. 505–525, 2020.
https://doi.org/10.1007/978-3-030-65411-5_25

Schwabe [35] argue why their circuit-based AES implementations are side-channel resistant.

Manually developing circuit-based implementations from algorithm specifications is costly and error-prone due to the huge gap between the two levels of abstraction. Moreover, to run software-based circuits in a real-world setting, additional code is needed, e.g., to initialize the inputs. Since this additional code is a potential source of leakage, its development requires a high level of rigor.

Unfortunately, there is currently no end-to-end tool support for this complex task: Existing tools for generating circuit-based crypto implementations require the input specification to be already at the level of a circuit description [9,48]. Conversely, existing tools for high-level synthesis that operate, e.g., on ANSI C or SystemC programs do not generate software. Instead, they transpile code to hardware description languages like Verilog or VHDL [51], from which logic synthesis tools (e.g., [62]) can derive FPGA configurations or ASIC designs.

We address this open problem by proposing an approach that hardens high-level C implementations by translating them into circuit-based software implementations. Our approach applies the hardening selectively, based on automatic quantitative program analysis. To support the translation to circuit format, our approach leverages existing compiler infrastructures from the area of secure computation, where circuit compilers, e.g., [15,32,40,43,61,72], are used to generate circuit descriptions that obliviously evaluate functions on private inputs via homomorphic encryption [27] or interactive cryptographic protocols [28,71].

We implement our approach in our toolchain RiCaSi, which takes as input a regular C implementation (e.g., of a block cipher) and outputs a circuit-based x86 binary together with a reliable quantitative security guarantee with respect to cache-side-channel leakage. Naturally, these security guarantees are based on established formal models. RiCaSi builds on the circuit compiler HyCC [15] and the program analysis tool CacheAudit [25], augmented with novel implementations and extensions required for the toolchain integration. Supplementary downloads are freely available at www.mais.informatik.tu-darmstadt.de/ricasi.html.

We evaluate RiCaSi across lookup-table-based AES implementations from the libraries OpenSSL [55], mbedTLS [7], Nettle [49] and LibTomCrypt [41], and across implementations of DES [52], 3DES, and Camellia [4] from mbedTLS. RiCaSi is easily applicable to all of these implementations. Moreover, it successfully improves their level of cache-side-channel security. For instance, the analysis integrated in RiCaSi derives an upper bound of 73.82 bit on the amount of information that the original OpenSSL AES might leak to an access-based cache side-channel attacker. After the conversion to a circuit-based implementation, this leakage bound drops to 0 bit. These upper bounds are based on rigorous program analysis and, hence, constitute reliable security guarantees.

Overall, we summarize our contributions as follows:

1. We present RiCaSi, a toolchain that semi-automatically produces circuit-based implementations of block ciphers with corresponding quantitative security guarantees on cache side-channel leakage.

2. We evaluate RiCaSi across implementations of AES, DES, 3DES, and Camellia, demonstrating the effectiveness of our approach by obtaining 0 bit upper leakage bounds for previously vulnerable implementations.
3. We furthermore evaluate the run-time and storage overhead induced by RiCaSi, demonstrating its practicality for security-critical applications.

2 Preliminaries

2.1 The Block Ciphers AES, DES and Camellia

AES. The Advanced Encryption Standard (AES) [53] is a block cipher that encrypts 128 bit message blocks using a symmetric secret key of size 128, 192, or 256 bit. To this end, AES creates so-called round keys from the secret key. The first round key is added to the message block using bitwise XOR. The remaining round keys are used to transform the block in multiple rounds.

The original AES proposal [20] suggests optimizing performance by precomputing the results of the transformation rounds for all possible inputs and storing them in lookup tables. Then, one simply needs to look up the transformation result from the table at the index corresponding to the current round input. The round inputs are the round key and the current state of the transformed message. Implementations that follow this table-based technique are prone to cache side-channel attacks: The indices of the table accesses and, hence, the addresses of the accessed memory locations depend on the secret message and round keys. If a memory entry is loaded into a cache that is shared with an attacker, the attacker might notice the presence of the entry in the cache and deduce secret information. He might even recover the entire secret key [2,5,6,11,31,36].

DES. The Data Encryption Standard (DES) [52] is a block cipher that encrypts 64 bit message blocks using a symmetric secret key of size 56 bit. Triple DES (3DES) is an extension for a key size of 168 bit, essentially performing three DES encryptions sequentially using three 64 bit substrings of the 3DES key as DES keys. Both DES and 3DES are deprecated [54], but they are still part of many common crypto libraries like mbedTLS [7] and OpenSSL [55].

Implementation of DES and 3DES might be susceptible to cache side-channel attacks. DES keys can, e.g., be recovered based on the cache misses encountered by implementations that use eight lookup tables (S-Boxes) for substitutions [67]. Such an implementation with eight lookup tables is, e.g., available in mbedTLS.

Camellia. Camellia [4] is a block cipher that encrypts 128 bit message blocks with symmetric secret 128, 192, or 256 bit keys in transformation rounds. Like AES and DES, Camellia uses round keys in each transformation round.

There are multiple techniques for cache attacks on implementations of Camellia that use lookup tables (S-Boxes). The Camellia secret key can, e.g., be recovered from an implementation with four tables using cache-access patterns obtained from power measurements [59]. Access-driven cache attacks can also be used to recover keys from a table-based Camellia implementation [73]. Table-based implementations are available, e.g., in OpenSSL [55] and mbedTLS [7].

2.2 Boolean Circuits for Secure Computation

Secure computation techniques make it possible to involve untrusted parties in the processing of private data. More specifically, homomorphic encryption allows one to outsource computation on private data to untrusted third parties [27]. In contrast, in secure two- or multi-party computation, two or more mutually distrusting parties jointly and interactively compute on private data [28,71].

Secure computation techniques obliviously compute publicly known functions expressed as combinatorial Boolean and/or arithmetic circuits. Boolean circuits are composed of AND and XOR gates, whereas arithmetic circuits consist of addition and multiplication gates. Both types of circuits are functionally complete when having access to constants, i.e., they can represent arbitrary computable functions. A Turing machine T with input length n can be expressed as a circuit of size $\tilde{O}(t(T, n))$, where $t(T, n)$ denotes the running time of T on input length n [30].

As noted in [26], the evaluation of Boolean and arithmetic circuits as done in secure computation is inherently secure against a wide range of software side-channel attacks. This is due to the fact that every possible branch of the function represented by such a circuit is executed in parallel and that the memory accesses performed by such circuit implementations do not depend on input data.

Unfortunately, designing circuits from high-level function descriptions is complex and requires tool support. Moreover, for secure computation, expert knowledge about the underlying protocols is required to achieve efficient results.

In hardware design, there exist academic as well as commercial high-level synthesis tools that automatically transpile, e.g., ANSI C or SystemC code to hardware description languages like Verilog or VHDL [51]. Via established logic synthesis tools (e.g., [62]) that output FPGA configurations or ASIC designs, it is furthermore possible to go to hardware level. Logic synthesis tools have also been adapted for secure computation by providing customized ASIC cell libraries and optimization parameters as well as algorithms [21,61,63,64].

Being much more convenient for regular software developers, a line of research has focused on creating optimized compilers that directly transform ANSI C programs to basic (Boolean) circuit representations that can easily be evaluated in software, e.g., by secure computation frameworks like ABY [22]. State-of-the-art in this domain is HyCC [15], the successor of the CBMC-GC compiler [32], which in turn is based on the bounded model checker CBMC [17].

HyCC provides optimizations like automated parallelization of concurrent code, logic minimization, loop unrolling, and minimization of the resulting circuits. However, in this work we target only size-optimized Boolean circuits and hence do not use the computationally expensive optimization steps of HyCC.

2.3 Program-Analysis Approach

To quantify the leakage of x86 binaries through cache side channels, we use a combination of information theory and abstract interpretation. This approach was first established in [38], later extended and then implemented in

the tool CacheAudit [25], of which multiple variants have been developed (e.g., [13,24,46]). We build on CacheAudit and extend it with support for additional language features where necessary. Below, we describe the underlying approach in more detail.

We model a cache side channel as a deterministic, discrete, memoryless channel from an input alphabet (random variable X) to an output alphabet (random variable Obs). The min-entropy $H_\infty(X) = -\log_2 \max_i\ p(x_i)$ of X captures the uncertainty an attacker has about the secret input if the probability for each input x_i is $p(x_i)$ [60]. The conditional min-entropy $H_\infty(X|Obs) = -\log_2 \Sigma_{j=1}^{|Obs|} p(obs_j) \cdot \max_i \frac{p(obs_j|x_i) \cdot p(x_i)}{p(obs_j)}$ captures the attacker's remaining uncertainty after observing the channel output, where output obs_j occurs for secret x_i with probability $p(obs_j|x_i)$ and occurs overall with probability $p(obs_j)$. The information that an output reveals about the input is modeled by the min-entropy leakage $H_\infty(X) - H_\infty(X|Obs)$, which is upper bounded by $\log_2|Obs|$ bit [39,60].

Let X be the set of possible secret inputs (secret key and message) and Obs be the possible observations of a cache-side-channel attacker. We compute cache side-channel leakage bounds as $\log_2|Obs^{\overline{\mathcal{D}}}|$, where $Obs^{\overline{\mathcal{D}}}$ is an overapproximation of the reachable observations Obs. The overapproximation makes the analysis feasible and is done using abstract interpretation [19]. More concretely, we overapproximate the actual possible execution states $\mathcal{D}$ by more abstract execution states $\overline{\mathcal{D}}$ and the actual semantics $\text{upd}_{\mathcal{D}} : \mathcal{D} \times \mathcal{I} \to \mathcal{D}$ of instruction set $\mathcal{I}$ by an abstract semantics $\text{upd}_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \mathcal{I} \to \overline{\mathcal{D}}$. We then compute the reachable abstract observations according to $\text{upd}_{\overline{\mathcal{D}}}$ and count the number of actual observations they represent. We take the logarithm to obtain the leakage bound $\log_2|Obs^{\overline{\mathcal{D}}}|$.

We consider four models of cache side-channel attackers, i.e., four variants of Obs: (1) Attackers under the model *acc* can deduce which memory entries are cached in a shared cache after the victim program is executed, (2) attackers under *accd* can deduce the number of memory entries the victim loaded into each cache set of such a shared cache, (3) attackers under *trace* can deduce the trace of cache hits and cache misses that occurred during the victim-program execution, and (4) attackers under *time* can deduce the execution time of a victim-program execution (modeled by fixed durations for cache hits, misses, and other steps).

3 The RiCaSi Toolchain

The goal of RiCaSi is to allow developers to obtain x86 binaries from regular and potentially vulnerable C code that come with quantitative security guarantees with respect to cache side channels. The high-level overview of our toolchain and its workflow is depicted in Fig. 1.

First, the C code provided by the user (e.g., a block-cipher implementation) is compiled to an x86 binary (e.g., with GCC) and analyzed with our extended version of CacheAudit (cf. Sect. 3.5). If the resulting upper bound on the cache-side-channel leakage of the binary is below an acceptable threshold, the binary can be used securely and RiCaSi terminates.

In case the threshold leakage is exceeded, RiCaSi compiles the C code into a circuit representation, which in turn is compiled into an x86 binary for further analysis. For this, we first preprocess the C code to substitute constructions currently not supported by existing circuit compilers (cf. Sect. 3.2), e.g., with respect to memory management and the passing of parameters.

The resulting C code is then compiled with the state-of-the-art circuit compiler HyCC [15] (cf. Sect. 3.3). For transforming the resulting circuit representation back to C code, we implement our own tool in Python (cf. Sect. 3.4).

After compilation to an x86 binary, we perform a second round of analysis with our extension of CacheAudit (cf. Sect. 3.5). Here, the expected output is an improved security guarantee in the form of an upper leakage bound that lies below the acceptable threshold or is even equal to 0 bit leakage.

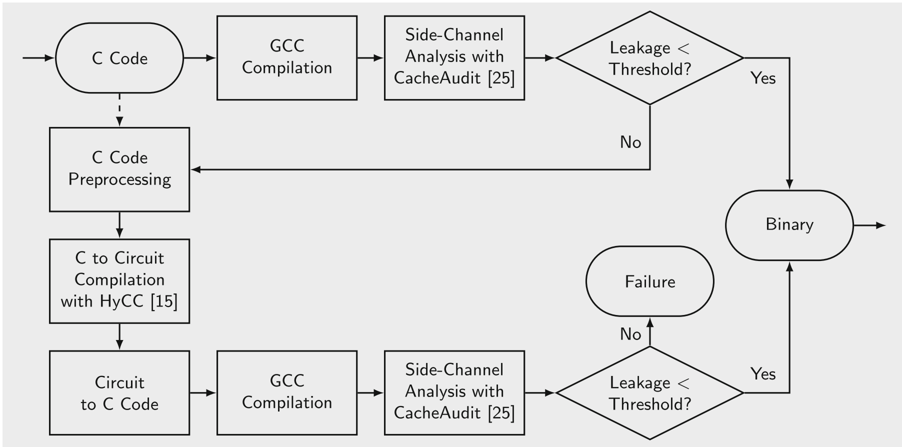

Fig. 1. Overview of our RiCaSi toolchain and workflow.

In the following, we detail the individual steps of the RiCaSi toolchain at the running example of a lookup-table-based implementation of AES encryption from OpenSSL (cf. Listing 1). The original implementation is vulnerable to cache side-channel attacks because it accesses lookup tables (e.g., table `Te0` as shown in Listing 1) at round-key dependent indices.

```
static const u32 Te0[256] = {0xc66363a5U, ...}; // lookup table
...
void AES_encrypt(const unsigned char *in, ...) {...
    s0 = GETU32(in) ^ rk[0]; ... // initial round
    t0 = Te0[s0 >> 24] ^ ...; // secret-dependent memory access
...}
```

Listing 1. Excerpt of OpenSSL AES encryption.

3.1 Initial Side-Channel Analysis

In the first step of RiCaSi, we apply automatic program analysis to determine whether the input implementation can be used as is or whether any hardening against cache side channels is required. To this end, we derive quantitative security guarantees for the x86 binary corresponding to the given implementation. More concretely, we compute upper bounds on the cache side-channel leakage of the binary and compare them to the threshold for the desired level of security. If non-zero bounds are not acceptable, the threshold can be set to zero. Technically, we use a combination of information theory and abstract interpretation, implemented in the tool CacheAudit. The tool takes an x86 binary and outputs bounds on the min-entropy leakage to the attacker models *acc*, *accd*, *trace* and *time* (cf. Sect. 2.3).

If the resulting leakage bounds lie below the desired leakage threshold, no hardening is required and unnecessary overhead can be avoided. If the leakage bounds are too high, we proceed with the preprocessing for circuit compilation.

Listing 2 shows the leakage bounds for OpenSSL AES from Listing 1. The bounds guarantee, for instance, that at most 73.83 bit are leaked to an attacker under the model *acc* (cf. Line 2 in Listing 2) and at most 70.34 bit are leaked to an attacker under *accd* (cf. Line 3). These bounds are rather weak security guarantees and would likely exceed the acceptable threshold leakage for most applications such that further steps of RiCaSi would be applied.

```
...
Number of valid cache config. (shared memory): ... (73.820808 bits)
Number of valid cache config. (disj. memory): ... (70.339850 bits)
# traces: ..., 280.000000 bits
# times: ..., 8.134426 bits
Analysis took 18 seconds.
```

Listing 2. Excerpt of analysis results for OpenSSL AES.

3.2 C Code Preprocessing

The "C-to-circuit" compilation as provided by HyCC [15] comes with several limitations regarding the processable source code. To avoid compilation issues, we manually apply several preprocessing steps to make existing implementations compatible. Although these steps are targeted towards our case studies (cf. Sect. 4), they might be of independent interest and worth to fully automate, as they can be applied to make the regular usage of HyCC more convenient.

1. Especially when compiling code that depends on an extensive library, it is best to first bundle all required methods in a single file. If possible, all method calls are replaced by inlining the required code into a method named `mpc_main`. Otherwise, debugging compilation errors becomes infeasible.
2. Global and static variables are not supported by HyCC. They must instead be declared in the main method (cf. the declaration of `Te0` in Listing 3).

3. Memory management via `malloc` and `calloc` is not supported by HyCC. Often, it is sufficient to declare arrays with fixed size instead. In many cases, it is also possible to simply remove such statements as the compiler can determine array sizes from later assignments. However, dealing with such memory management issues was not required for any of our case studies.
   HyCC also does not support passing arrays or pointers in method headers. This can be circumvented by splitting arrays into single variables, which are passed instead (cf. passing the plaintext data `in01` in Listing 3).

The preprocessed OpenSSL AES encryption code is shown in Listing 3.

```
int mpc_main(unsigned char in01, ...) {... // inputs split in bytes
  const unsigned char in[16] = {in01, ...}; // reconstruct inputs
  const u32 Te0[256] = {0xc66363a5U, ...}; // table declaration
...}
```

Listing 3. Excerpt of preprocessed OpenSSL AES encryption.

3.3 C Code to Circuit Compilation

The compilation of the preprocessed C code to a circuit description happens through a straightforward application of HyCC [15]. For the compilation, the C code is first transformed into a "goto code" intermediate representation, loops are unrolled, variables are split into single bits, and operations over those bits are expressed as Boolean functions [32]. As briefly described in Sect. 2.2, HyCC also performs several optimizations like circuit minimization.

However, in comparison to the regular usage of HyCC, several of the most computationally expensive steps can be skipped. This is because here we target only the creation of size-optimized Boolean circuits and do not consider depth-optimized Boolean or arithmetic circuits (which are beneficial for some interactive secure computation protocols [28]).

Therefore, for our purpose, HyCC does not need to decompose the code into separate modules and compile each module into multiple different types of circuits. We can also skip the final step where HyCC tries to heuristically optimize the total cost for secure computation protocols by finding the best possible combination of different types of circuits and protocols.

3.4 Circuit to Binary Compilation

In the following, we describe how the HyCC circuit output is translated into C code and further compiled into an x86 binary.

The circuit output produced by HyCC is by default in a binary format to be processed by the ABY MPC framework [22] via a specialized adapter. To facilitate further processing without ABY, we use a HyCC export functionality for conversion into the human-readable and widely used BRISTOL circuit format [66]. In the BRISTOL format, every line of the circuit description

file declares the type of one gate as well as the number and the identifiers of the gate's input and output wires. The supported gate types are AND, XOR, and INV (inversion). The header of the circuit description specifies the total number of gates, the total number of wires, and circuit input as well as output wires. The HyCC output in the BRISTOL format for OpenSSL AES is shown in Listing 4.

```
490425 490809
384 0 160
...
2 1 121 377 385 XOR // XOR gate in BRISTOL representation
1 1 385 386 INV     // INV gate in BRISTOL representation
2 1 384 386 387 AND // AND gate in BRISTOL representation
...
```

Listing 4. HyCC circuit for OpenSSL AES encryption in BRISTOL format.

We implemented a converter tool in Python to translate BRISTOL circuit description files into C source code. The converter is controlled via a configuration file that, besides the circuit name and file, specifies input and output types, and which external libraries (e.g., `stdio.h`) should be included.

The converter first declares a variable for each wire and disassembles the specified inputs into the respective circuit input wires. It then iterates through each line of the circuit description and inserts the respective C instruction for performing the specified gate operation (e.g., `&` for AND gates) on the variables corresponding to the gate input and output wires. This is possible because the gates in the BRISTOL circuit format are ordered topologically, i.e., all input wires for each gate have been assigned before. Finally, the circuit output is assembled in the configured type from the circuit output wires. The converter output for our OpenSSL AES encryption example is shown in Listing 5.

```
int openssl_aes_enc(int in01, ...) {
  unsigned char w0, ..., w490808;
  int inbits01[8] = split(in01);
  w0 = inbits01[0];
  ...
  w385 = w121 ^ w377; // C code for gate 2 1 121 377 385 XOR
  w386 = !w385;       // C code for gate 1 1 385 386 INV
  w387 = w384 & w386; // C code for gate 2 1 384 386 387 AND
  ...
}
```

Listing 5. Excerpt of OpenSSL AES encryption in circuit-style C.

The resulting C code file can then be compiled into an x86 binary using, for example, the GCC compiler. It is also possible to integrate the produced C code with another application before compilation, or to modify the code, e.g., to include further input and output processing.

3.5 Final Side-Channel Analysis

To ensure that no potential for cache side-channel leakage remains in the final circuit binary, we perform an additional program analysis step. To this end, we apply a variant of the tool CacheAudit that we extended for the purpose of analyzing circuit binaries. Our variant of CacheAudit augments the prior version in two directions: support for large control-flow graphs and support for additional x86 opcodes.

Circuit-based binaries are significantly larger than regular binaries because all individual gates are encoded in the assembly code. Since CacheAudit was not intended for the analysis of binaries with large basic blocks, its parser quickly runs into overflows when trying to build a control-flow graph for the studied circuit-based binaries. By rewriting the corresponding parts of the CacheAudit implementation in a tail-recursive style, we now avoid this issue.

Furthermore, circuit-based binaries use x86 opcodes that did not occur in the binaries that have been analyzed with CacheAudit before. In particular, the comparison instructions with opcodes `0xA8` and `0xF7/0` occur in the binaries. We added support for both instructions to CacheAudit.

Our resulting variant of CacheAudit can be successfully applied to all circuit-based binaries in our evaluation and is of independent interest.

Listing 6 shows an excerpt of the analysis results for the x86 binary corresponding to the circuit-compiled variant of OpenSSL AES from Listing 5. In this example, the resulting leakage bounds are 0 bit across all four attacker models (cf. Line 2 in Listing 6 for *acc*, Line 3 for *accd*, Line 4 for *trace* and Line 5 for *time*). That is, the circuit-compiled binary does not leak secret information through cache side channels to attackers under any of these attacker models.

```
...
Number of valid cache config. (shared memory): 1, (0.000000 bits)
Number of valid cache config. (disj. memory): 1, (0.000000 bits)
# traces: 1, 0.000000 bits
# times: 1.000000, 0.000000 bits
Analysis took 185392 seconds.
```

Listing 6. Excerpt of analysis results for circuit-compiled OpenSSL AES.

Note that circuit compilation does not inevitably lead to 0 bit leakage bounds. Since side channels are vulnerabilities at the level of implementation details, it is crucial to ensure that the hardening is effective in all details. In an intermediate version of RiCaSi we had accidentally introduced potential side-channel leakage in the circuit-to-C compilation step: our initialization of the circuit inputs was not constant-time. With the final program-analysis step of RiCaSi, we detected the mistake due to unexpectedly high leakage bounds for the generated binary. We then adapted our circuit-to-C compilation tool accordingly. As shown in Listing 6, the hardening with RiCaSi is now effective in all details, leading to 0 bit leakage bounds for the resulting binary.

4 Evaluation of Cache-Side-Channel Security

We evaluate the applicability of RiCaSi and the benefit it provides in terms of cache side-channel security guarantees in two dimensions.

We first consider a range of lookup-table-based AES implementations: an implementation from OpenSSL [55] that uses four lookup tables of size 1 kB, an implementation from mbedTLS [7] (a library used, e.g., by cURL [65] and OpenVPN [56]) that uses four 1 kB tables and a 0.25 kB S-Box, an alternative implementation with lookup tables and an S-Box from Nettle [49], and one implementation from the library LibTomCrypt [41] that uses eight 1 kB lookup tables.

In the second step, we broaden the evaluation to implementations of other block ciphers. We consider implementations of three additional block ciphers from the library mbedTLS: Camellia, DES, and 3DES.

4.1 RiCaSi for AES Implementations

We analyze the sequence of the key-generation and encryption functions from the respective AES implementations, applied to a 256 bit key and 128 bit plaintext. We configure mbedTLS without x86 VIA PadLock instructions because we are interested only in the software AES implementation. We configure LibTomCrypt to omit assert statements with indirect jumps to make the computation of security guarantees with state-of-the-art program analysis feasible. The details on the configurations that we used are summarized in Table 1.

Table 1. AES implementations inspected in our case study.

Library	Version	Configuration	Analyzed functions
OpenSSL	1.1.1d	default	`AES_set_encrypt_key`, `AES_encrypt`
mbedTLS	2.16.5	removed `MBEDTLS_PADLOCK_C`	`mbedtls_aes_init`, `mbedtls_aes_setkey_enc`, `mbedtls_aes_encrypt`, `mbedtls_aes_free`
Nettle	3.5	default	`aes256_set_encrypt_key`, `aes256_encrypt`
LibTomCrypt	1.18.2	`ARGTYPE`	`rijndael_enc_setup`, `rijndael_enc_ecb_encrypt`

To compute guarantees for the cache side-channel security of these implementations before and after circuit compilation, we use our extension of the program analysis tool CacheAudit as described in Sect. 3.5.

Our analysis with CacheAudit and the resulting security guarantees focus on one single level of cache. This is a common simplification of modern multilevel cache hierarchies that is frequently applied in cache side-channel quantification [13,24,25,46]. In our analysis, we consider an 8-way set associative

cache with 64 cache sets and a line size of 64 Bytes with the PLRU cache line replacement strategy. This reflects, e.g., the L1 data caches of the Intel Skylake architecture [33, Table 2–4], [1] and the AMD Zen2 architecture [3].

As a baseline for our evaluation, we compute upper bounds on the cache side-channel leakage of the original, vulnerable AES implementations. We then harden the implementations using RiCaSi. In the final analysis step of RiCaSi, CacheAudit is applied again to compute cache-side-channel leakage bounds for the hardened implementations. In both cases, we consider the 32 bit x86 binaries obtained from the C implementations using `gcc` version `5.4.0`.

Baseline Results. Our baseline analysis results across the AES implementations and side-channel attacker models described in Sect. 3.5 are shown on the left side of Table 2. We round the leakage bounds to two decimal places.

Table 2. AES leakage bounds in [bit] before (left) and after (right) RiCaSi.[a]

Cipher	Attacker Model			
	acc	*accd*	*trace*	*time*
OpenSSL	73.82	70.34	280.00	8.13
mbedTLS	88.09	81.55	287.00	8.17
Nettle	85.93	78.55	299.00	8.23
LibTomCrypt	204.03	143.43	274.00	8.10

Cipher	Attacker Model			
	acc	*accd*	*trace*	*time*
OpenSSL	0.00	0.00	0.00	0.00
mbedTLS	0.00	0.00	0.00	0.00
Nettle	0.00	0.00	0.00	0.00
LibTomCrypt	0.00	0.00	0.00	0.00

[a]For homogeneity across tables, we use the full display format also for all-zero tables.

CacheAudit yields rather high leakage bounds, between 70.34 bit and 299.00 bit for the attacker models *acc*, *accd* and *trace* across all libraries. For the attacker model *time*, the bounds are lower and lie between 8.10 bit and 8.23 bit. For instance, the *time* leakage bound for OpenSSL AES is 8.13 bit. This means that one execution of this AES binary leaks at most 2.12% of the 384 secret bits (256 bit key and 128 bit plaintext) to an attacker under the model *time*.

The high leakage bounds for *acc*, *accd* and *trace* are rather weak security guarantees. That is, for the attacker models *acc*, *accd* and *trace*, the level of security on which we can rely is rather low. Even for the attacker model *time*, we do not obtain guarantees for the complete absence of leakage.

The high bounds are not surprising because all analyzed binaries belong to lookup-table-based implementations that use secret-dependent memory accesses. Next, we evaluate how effective RiCaSi is in hardening the implementations.

Results for RiCaSi. The leakage bounds for the circuit-based binaries produced by RiCaSi are shown on the right-hand side of Table 2. Note that the upper bounds on the leakage are 0 bit across all implementations and attacker models. That is, no information is leaked to attackers under the four models.

While the 0 bit leakage bounds might not be surprising at first sight, recall that they play a central role in RiCaSi. If any detail of the circuit compilation and translation failed, as in our prior implementation (cf. Sect. 3.5), we would

spot this here. With 0 bit bounds, we can be sure that the hardening with RiCaSi is effective in all implementation details.

Table 3. DES, 3DES and Camellia implementation inspected in our case study.

Cipher	Key length	Plaintext length	Analyzed functions
Camellia	256 bit	128 bit	`mbedtls_camellia_init`, `mbedtls_camellia_setkey_enc`, `mbedtls_camellia_crypt_ecb`, `mbedtls_camellia_free`
DES	64 bit	64 bit	`mbedtls_des_init`, `mbedtls_des_setkey_enc`, `mbedtls_des_crypt_ecb`, `mbedtls_des_free`
3DES	128 bit	64 bit	`mbedtls_des3_init`, `mbedtls_des3_set2key_enc`, `mbedtls_des3_crypt_ecb`, `mbedtls_des3_free`

4.2 RiCaSi for Block Ciphers from mbedTLS

For each of the three block ciphers Camellia, DES, and 3DES, we analyze the respective sequence of functions to initialize the data structures, compute the key schedule, perform the encryption and free the data structures from mbedTLS version `2.16.5`. The details, including key and plaintext lengths, are described in Table 3. We use the same CacheAudit variant and configuration as in Sect. 4.1.

Table 4. mbedTLS leakage bounds in [bit] before (left) and after (right) RiCaSi.

Cipher	Attacker Model			
	acc	*accd*	*trace*	*time*
AES	88.09	81.55	287.00	8.17
Camellia	28.50	25.75	242.00	7.92
DES	38.40	37.75	141.00	7.16
3DES	52.20	48.34	416.00	8.70

Cipher	Attacker Model			
	acc	*accd*	*trace*	*time*
AES	0.00	0.00	0.00	0.00
Camellia	0.00	0.00	0.00	0.00
DES	0.00	0.00	0.00	0.00
3DES	0.00	0.00	0.00	0.00

Baseline Results. The leakage bounds for the original block-cipher implementations from mbedTLS (including mbedTLS AES for comparison) are shown on the left side of Table 4. The *acc* and *accd* leakage bounds for Camellia, DES, and 3DES are lower than the leakage bounds for AES, but still rather high compared to the respective sizes of the secret key and message (384 bit for Camellia,

128 bit for DES, and 192 bit for 3DES). For the attacker models *trace* and *time*, the leakage bounds for 3DES are even higher than those for AES. This might be due to an accumulation of leakage across the DES executions in 3DES.

Again, the high leakage bounds are not surprising given the known cache-side-channel attacks on such implementations described in Sect. 2. Next, we apply RiCaSi to harden the implementations against such attacks. The resulting leakage bounds are shown on the right-hand side of Table 4.

Results for RiCaSi. For all block-cipher implementations hardened with RiCaSi, we are able to derive guarantees for 0 bit leakage for all four cache side-channel attacker models. That is, RiCaSi effectively hardened the implementations against cache side-channel attackers under these models.

Overall, RiCaSi hence supports the hardening not only of AES implementations but also of a broader range of block-cipher implementations. In all cases that we considered in our evaluation, the effectiveness of the hardening was automatically verifiable using the program analysis of CacheAudit.

5 Evaluation of Overhead

Compiling applications into side-channel resistant executables is a one time cost that is quickly amortized over time. However, RiCaSi generates repeated overhead in two aspects, which we evaluate in detail: First, we study how much the size of the circuit-based binaries increases compared to regular compilation results. Then, we evaluate how much the run-time of the side-channel resistant binaries increases compared to the vulnerable counterparts.

5.1 Binary Sizes

In Table 5, we compare the binary sizes of the regular block-cipher implementations to the output produced by RiCaSi. While storage costs nowadays are almost negligible at the given scale, considering the overhead in terms of binary sizes is especially necessary to estimate the additional costs when widely distributing software over the Internet, or for embedded devices.

The results in Table 5 strongly vary among the considered block ciphers. The binary sizes for DES and 3DES, e.g., stay well below 5 MB and have less than factor 5× blow-up. However, binary sizes for AES increase up to about 24 MB, which corresponds to a blow-up of two orders of magnitude. Therefore, we recommend to use RiCaSi mainly on small, highly security critical code sections.

Note that the compilation setup in our case studies was not tailored to optimize the binary sizes. All binaries include debug information. Moreover, while the original AES binaries were linked dynamically, we used static linking for all other binaries to make them self-contained for the program analysis. By dropping dispensable information from the binaries, the sizes could be reduced if necessary.

Table 5. Comparison of binary sizes.

Cipher	Library	Original (in KB)	RiCaSi (in KB)	Overhead
AES	OpenSSL	37.72	23,624.18	626.30×
	Nettle	29.81	23,573.93	790.81×
	LibTomCrypt	56.70	23,623.09	416.63×
	mbedTLS	57.32	23,581.20	411.40×
Camellia	mbedTLS	890.56	11,923.80	13.39×
DES	mbedTLS	891.80	1,408.01	1.58×
3DES	mbedTLS	891.84	3,497.00	3.92×

5.2 Run-Times

We evaluate the run-times of the executables of various block ciphers generated by RiCaSi and compare the resulting overhead to the regular vulnerable executables in Fig. 2. All binaries are executed on one logical core of an Intel Core i9-7960X CPU clocked at 2.8 GHz (with up to 4.2 GHz turbo boost). The stated run-times are averages over 10 executions.

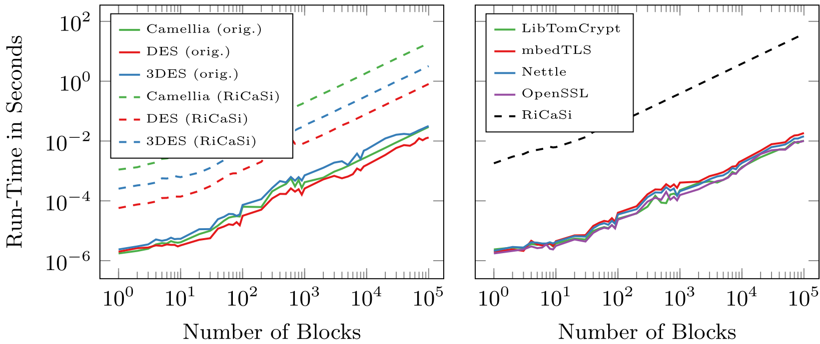

Fig. 2. Comparison of run-times for encrypting an increasing number of blocks with different ciphers. Left: Camellia, DES, 3DES (mbedTLS). Right: different AES implementations; the differences in the respective RiCaSi versions are negligible.

In Fig. 2, we observe about two (DES, 3DES) to three (Camellia, all AES implementations) orders of magnitude overhead when executing the binaries produced by RiCaSi. For encrypting large amounts of data (i.e., in the order of gigabytes) or applications with strict real-time requirements (e.g., Bitlocker), this overhead quickly becomes impractical. However, for processing small or even large amounts of data in high-security contexts without strict real-time requirements, the binaries generated by RiCaSi deliver practical performance.

6 Related Work

6.1 Secure Computation Techniques for Side-Channel Mitigation

So-called one-time programs (OTPs) are studied in [30], which are programs that can be evaluated only on a single input chosen at run-time. The proposed construction is based on a combination of tamper-resistant hardware with Yao's garbling scheme for Boolean circuits [71]. In this scheme, the gate tables are encrypted and the corresponding keys required for decryption are carried by the circuit wires instead of single bits. Importantly, the nature of the garbled circuit evaluation prevents all potential side-channel leakage.

A variant of this idea was later implemented on FPGAs by [34]. Their performance evaluation observes an overhead of about factor $10^6\times$ comparing one unprotected AES evaluation in hardware to a provably side-channel resistant hardware-accelerated OTP evaluation. Despite this significant overhead, the authors argue their solution might be reasonable for high-security applications.

In contrast to these works, we provide a generic compiler toolchain for creating and evaluating Boolean (non-garbled) circuits in software. Our performance evaluation shows an overhead of only about factor $10^3\times$ when comparing regular vulnerable implementations of various block ciphers to circuit-based executables with 0 bit upper leakage bounds with respect to cache side channels.

In [26], Felsen et al. use circuit representations to mitigate side-channel attacks for programs shielded with Intel Software Guard Extensions (SGX) [18]. Intel SGX is a trusted execution environment available in many Intel CPUs that allows one to run so-called *enclaves* in isolation from all other software. However, Intel considers software side channels out of scope for the attacker model, which resulted in many attacks showing the vulnerability especially of enclaves running cryptographic code (e.g., [69]). As a solution, Felsen et al. created an enclave that evaluates Boolean circuits on private inputs provided to the enclave via secure channels [26]. They claim resistance against timing and page-table- as well as cache-based software side channels, but do not provide any analyses for confirmation. Also, they do not provide an integrated solution for obtaining the circuit representations required for their circuit evaluator.

In contrast to [26], we provide an automated way to generate side-channel resistant executables with our toolchain RiCaSi. Most importantly, our approach is backed by formal analyses showing upper bounds of 0 bit on the cache side-channel leakage for various implementations of block ciphers. In the future, RiCaSi could be extended to produce side-channel resistant Intel SGX enclaves.

The concept of Oblivious RAM (ORAM) [29] was introduced to prevent that code can be reverse-engineered from observations about the memory accesses performed by the code. The key idea is to replace each memory access with a sequence of memory accesses that conceals the address of the original memory access. That is, ORAM prevents information leakage via memory accesses without removing these accesses completely. RiCaSi follows the alternative approach of eliminating the memory accesses through circuit compilation.

6.2 Systematic Detection and Assessment of Side-Channel Leakage

Systematic approaches to side-channel security range from qualitative approaches, like type-based techniques [8,23,37,50], to quantitative approaches, like abstraction-based techniques [25,38,42] or experiment-based techniques [16,44,45]. In the following, we provide an overview of existing qualitative and quantitative approaches with a focus on cache side channels.

Qualitative approaches to cache side-channel detection include, e.g., DATA [70] and CacheD [68]. Both tools check for cache side channels in execution traces. They are intended for debugging and do not provide security guarantees. The tool CaSym [14] soundly verifies LLVM code against cache side channels. While DATA uses statistical methods, CacheD and CaSym use symbolic execution.

Quantitative approaches to cache side-channel assessment include multiple variants of the tool CacheAduit [25]. CacheAudit computes upper bounds on the cache side-channel leakage of x86 binaries using a combination of information theory and abstract interpretation. It has been successfully extended and applied for the analysis of multiple cryptographic implementations, including AES implementations [46], modular exponentiation [24], and lattice-based cryptography [13]. Our work is based on CacheAudit and extends the tool for our purposes with better scalability and additional x86 language coverage.

6.3 Analysis of Side-Channel Leakage in Circuit Implementations

To the best of our knowledge, the closest to our work in combining circuit compilation with side-channel security guarantees are the Usuba compiler [48] and its extension to Tornado [9]. Both, Usuba and Tornado take as input a circuit specification in the Usuba specification language.

Usuba compiles the specification to C code and introduces optimizations like bitslicing. Bitslicing [12] optimizes the performance of circuit-based software implementations by parallelization. To this end, the variables that model the circuit wires are used to store multiple bits instead of just one bit. Applying bitwise operations to these variables will then model the application of the corresponding gate to all bits in parallel.

Tornado augments Usuba and returns an optimized circuit binary that satisfies security guarantees with respect to the register-probing adversary model. That is, the resulting circuit is secure against side-channel adversaries that can probe intermediate values of registers during the execution of the software circuit. To this end, Tornado extends Usuba with support for the masking countermeasure. Masking mitigates side-channel leakage by splitting the secret value into shares that are only meaningful in combination. Moreover, Tornado combines the extended Usuba with the tool TightProve [10] to show that the resulting masked implementation is secure with respect to the register-probing model.

That is, both Usuba and Tornado work at the level of circuits. Both optimize the circuits and Tornado also provides security guarantees. Neither of the tools aims at supporting the development of circuits from high-level specifications.

Overall, Tornado and Usuba are complementary to RiCaSi. Tornado and Usuba focus on optimizing circuits, e.g., by bitslicing. RiCaSi currently uses only one bit of each variable, i.e., does not apply bitslicing. Tornado and Usuba do not support the generation of a circuit specification from a high-level implementation. RiCaSi closes this gap and converts high-level C implementations into circuit-based implementations that are reliably secure against cache side-channel attacks.

7 Conclusion

In this paper, we presented the toolchain RiCaSi, an integrated solution for hardening regular C implementations against cache side channels by transforming them into circuit-based x86 binaries.

RiCaSi applies program analysis to quantify the threat of cache side-channel leakage in a given implementation. Based on the analysis results, the implementation can be hardened selectively and unnecessary costs are avoided. With RiCaSi, we successfully transformed multiple vulnerable crypto implementations (AES from OpenSSL, mbedTLS, Nettle, and LibTomCrypt; Camellia, DES, and 3DES from mbedTLS) into circuit-based binaries with zero-leakage guarantees against four cache attacker models. For these binaries, we observed overhead of up to three orders of magnitude, which is acceptable for critical applications without hard real-time requirements. Overall, RiCaSi performs a selective, effective and affordable hardening of regular C implementations against cache side channels.

In the future, integrating steps for circuit optimization into the toolchain, e.g., the use of vectorized instructions or automated bitslicing as in [48], will be a promising direction to greatly reduce overhead while maintaining the applicability to high-level C implementations and the reliable security guarantees.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This project was co-funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. It has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI).

References

1. Abel, A., Reineke, J.: nanoBench: a low-overhead tool for running microbenchmarks on x86 systems. CoRR abs/1911.03282 (2019)
2. Aciiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: ICICS (2006)
3. Advanced Micro Devices: software optimization guide for AMD family 17h models 30h and greater processors. Publication number: 56305, Revision: 3.02 (2020)
4. Aoki, K., et al.: Specification of Camellia - a 128-bit block cipher, version 2.0 (2001)

5. Apecechea, G.I., Eisenbarth, T., Sunar, B.: S$a: a shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In: S& P (2015)
6. Apecechea, G.I., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! A fast, cross-vm attack on AES. In: RAID (2014)
7. ARM Limited: mbedTLS (Version 2.16.5) (2020). https://tls.mbed.org/download/start/mbedtls-2.16.5-apache.tgz
8. Barthe, G., Rezk, T., Warnier, M.: Preventing timing leaks through transactional branching instructions. In: QAPL (2006)
9. Belaïd, S., Dagand, P., Mercadier, D., Rivain, M., Wintersdorff, R.: Tornado: automatic generation of probing-secure masked bitsliced implementations. In: EUROCRYPT (2020)
10. Belaïd, S., Goudarzi, D., Rivain, M.: Tight private circuits: achieving probing security with the least refreshing. In: ASIACRYPT (2018)
11. Bernstein, D.J.: Cache-timing attacks on AES. University of Illinois at Chicago, Technical report (2005)
12. Biham, E.: A fast new DES implementation in software. In: FSE (1997)
13. Bindel, N., Buchmann, J.A., Krämer, J., Mantel, H., Schickel, J., Weber, A.: Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In: FPS (2017)
14. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.T.: Casym: cache aware symbolic execution for side channel detection and mitigation. In: S&P (2019)
15. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., Schneider, T.: HyCC: compilation of hybrid protocols for practical secure computation. In: CCS (2018)
16. Chothia, T., Kawamoto, Y., Novakovic, C.: A tool for estimating information leakage. In: CAV (2013)
17. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004)
18. Costan, V., Devadas, S.: Intel SGX explained. ePrint 2016/86 (2016)
19. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
20. Daemen, J., Rijmen, V.: AES submission document on Rijndael. Version **2** (1999)
21. Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S.: Automated synthesis of optimized circuits for secure computation. In: CCS (2015)
22. Demmler, D., Schneider, T., Zohner, M.: ABY - a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
23. Dewald, F., Mantel, H., Weber, A.: AVR processors as a platform for language-based security. In: ESORICS (2017)
24. Doychev, G., Köpf, B.: Rigorous analysis of software countermeasures against cache attacks. In: PLDI (2017)
25. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: a tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. **18**(1) (2015)
26. Felsen, S., Kiss, Á., Schneider, T., Weinert, C.: Secure and private function evaluation with Intel SGX. In: CCSW (2019)
27. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC (2009)
28. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC (1987)
29. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)

30. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: CRYPTO (2008)
31. Gullasch, D., Bangerter, E., Krenn, S.: Cache games - bringing access-based cache attacks on AES to practice. In: S&P (2011)
32. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: CCS (2012)
33. Corporation, Intel: Intel® 64 and IA-32 architectures optimization reference manual. Order Number **248966–032** (2016)
34. Järvinen, K., Kolesnikov, V., Sadeghi, A., Schneider, T.: Garbled circuits for leakage-resilience: hardware implementation and evaluation of one-time programs. In: CHES (2010)
35. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: CHES (2009)
36. Kim, T., Peinado, M., Mainar-Ruiz, G.: STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security (2012)
37. Köpf, B., Mantel, H.: Transformational typing and unification for automatically correcting insecure programs. IJIS **6**(2–3) (2007)
38. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: CAV (2012)
39. Köpf, B., Smith, G.: Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In: CSF (2010)
40. Kreuter, B., Shelat, A., Mood, B., Butler, K.R.B.: PCF: a portable circuit format for scalable two-party secure computation. In: USENIX Security (2013)
41. libtom projects: LibTomCrypt (Version 1.18.2) (2018). https://github.com/libtom/libtomcrypt/releases/tag/v1.18.2
42. Malacaria, P., Khouzani, M., Pasareanu, C.S., Phan, Q., Luckow, K.S.: Symbolic side-channel analysis for probabilistic programs. In: CSF (2018)
43. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: USENIX Security (2004)
44. Mantel, H., Schickel, J., Weber, A., Weber, F.: How secure is green it? the case of software-based energy side channels. In: ESORICS (2018)
45. Mantel, H., Starostin, A.: Transforming out timing leaks, more or less. In: ESORICS (2015)
46. Mantel, H., Weber, A., Köpf, B.: A systematic study of cache side channels across AES implementations. In: ESSoS (2017)
47. Matsui, M., Nakajima, J.: On the power of bitslice implementation on Intel Core2 processor. In: CHES (2007)
48. Mercadier, D., Dagand, P.: Usuba: high-throughput and constant-time ciphers, by construction. In: PLDI, pp. 157–173 (2019)
49. Möller, N.: Nettle (Version 3.5) (2019). https://ftp.gnu.org/gnu/nettle/nettle-3.5.tar.gz
50. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: automatic detection and removal of control-flow side channel attacks. In: ICISC (2006)
51. Nane, R., et al.: A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. CAD Integrat. Circ. Syst. **35**(10), 1591–1604 (2016)
52. National Institute of Standards and Technology: FIPS PUB 46–3: Data encryption standard (DES) (1999)
53. National Institute of Standards and Technology: FIPS PUB 197: Advanced encryption standard (AES) (2001)

54. National Institute of Standards and Technology: Update to current use and deprecation of TDEA (2017). https://csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA
55. OpenSSL Software Foundation: OpenSSL (Version 1.0.1d) (2020). https://www.openssl.org/source/openssl-1.0.1d.tar.gz
56. OpenVPN Inc: OpenVPN (2020). https://openvpn.net/
57. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: CT-RSA (2006)
58. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. ePrint 2002/169 (2002)
59. Poddar, R., Datta, A., Rebeiro, C.: A cache trace attack on Camellia. In: InfoSecHiComNet (2011)
60. Smith, G.: On the foundations of quantitative information flow. In: FoSSaCS (2009)
61. Songhori, E.M., Hussain, S.U., Sadeghi, A., Schneider, T., Koushanfar, F.: Tinygarble: highly compressed and scalable sequential garbled circuits. In: S&P (2015)
62. Synopsis: DC Ultra (2020). https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html
63. Testa, E., Soeken, M., Amarù, L.G., Micheli, G.D.: Reducing the multiplicative complexity in logic networks for cryptography and security applications. In: DAC (2019)
64. Testa, E., Soeken, M., Riener, H., Amaru, L., Micheli, G.D.: A logic synthesis toolbox for reducing the multiplicative complexity in logic networks. In: DATE (2020)
65. The cURL Team: cURL (2020). https://curl.haxx.se/
66. Tillich, S., Smart, N.: (Bristol Format) Circuits of basic functions suitable for MPC and FHE (2020). https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html
67. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: CHES (2003)
68. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: Cached: identifying cache-based timing channels in production software. In: USENIX Security (2017)
69. Weiser, S., Spreitzer, R., Bodner, L.: Single trace attack against RSA key generation in Intel SGX SSL. In: ASIACCS (2018)
70. Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., Sigl, G.: DATA - differential address trace analysis: Finding address-based side-channels in binaries. In: USENIX Security (2018)
71. Yao, A.C.: How to generate and exchange secrets (extended abstract). In: FOCS (1986)
72. Zahur, S., Evans, D.: Obliv-C: a language for extensible data-oblivious computation. ePrint 2015/1153 (2015)
73. Zhao, X., Wang, T., Zheng, Y.: Cache timing attacks on Camellia block cipher. ePrint 2009/354 (2009)