HyCaMi: High-Level Synthesis for Cache Side-Channel Mitigation*

Heiko Mantel
mantel@mais.informatik.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Joachim Schmidt
joachim.schmidt@stud.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Thomas Schneider
schneider@encrypto.cs.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Maximilian Stillger
maximilian.stillger@stud.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Tim Weißmantel
weissmantel@mais.informatik.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Hossein Yalame
yalame@encrypto.cs.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

ABSTRACT

Cache side-channels are a major threat to cryptographic implementations, particularly block ciphers. Traditional manual hardening methods transform block ciphers into Boolean circuits, a practice refined since the late 90s. The only existing automatic approach based on Boolean circuits achieves security but suffers from performance issues. This paper examines the use of Lookup Tables (LUTs) for automatic hardening of block ciphers against cache side-channel attacks. We present a novel method combining LUT-based synthesis with quantitative static analysis in our HyCaMi framework. Applied to seven block cipher implementations, HyCaMi shows significant improvement in efficiency, being 9.5× more efficient than previous methods, while effectively protecting against cache side-channel attacks. Additionally, *for the first time*, we explore balancing speed with security by adjusting LUT sizes, providing faster performance with slightly reduced leakage guarantees, suitable for scenarios where absolute security and speed must be balanced.

1 INTRODUCTION

Cache side-channels are unintended flows of information across system layers. In a cache side-channel attack, an adversary obtains information on secret inputs by observing the minute timing differences caused by cache hits and misses triggered by runs of the target program. Despite well-known countermeasures, prevalent cryptographic libraries like *OpenSSL*, *mbedTLS*, and *Nettle* often rely on implementations of AES-256 that are susceptible to such attacks, especially in the absence of specialized hardware like AES-NI.

In response to these vulnerabilities, developers have historically employed constant-time programming methods to manually harden cryptographic implementations. This approach is evident in the creation of bitsliced versions of DES [5] and AES-GCM [17], which are acknowledged for both their resistance to side-channel attacks and high performance. These secure versions are crafted using hand-designed Boolean circuits that inherently possess constant-time properties. However, this manual process is not foolproof and is susceptible to introducing new vulnerabilities, either through human error or unforeseen compiler optimizations. For instance, [9] has identified leaks in constant-time code running in production.

The pursuit of automatic side-channel hardening tools has become a focal point in recent research. Projects like Raccoon [28], SC-Eliminator [34], and Constantine [6] represent significant strides in this direction, as outlined in our literature review (see Table 1). However, while these tools make compelling cases for their effectiveness in enhancing side-channel security, they often lack comprehensive static verification of the security properties in the resulting binaries. In contrast *RiCaSi* [22] is more closely aligned with our work, as it utilizes a methodology similar to the initial manual hardening approach. It automatically generates Boolean circuits from C implementations, akin to the early manual methods, then translates them back to C and compiles to x86. This process allows for static analysis verification, ensuring that the resulting binaries are free from cache side-channels. However, a critical limitation of *RiCaSi* is the significant performance degradation it introduces, as the binaries generated are considerably slower compared to their original versions, highlighting a trade-off between security and efficiency.

Our research addresses this performance-security trade-off by adopting a shift similar to that seen in Multi-Party Computation (MPC): from Boolean circuits to Lookup Tables (LUTs). LUTs have been recognized as essential components in various computational areas, including cryptography and secure computation, especially in MPC, where LUT-based techniques have overwhelmingly been preferred over traditional Boolean circuit evaluations [7, 10, 26]. Our work aims to provide an automatic side-channel hardening solution that not only retains relative speed but also offers robust, statically verifiable security guarantees through the use of LUTs.

1.1 Related Work

This section provides a concise overview of related works.

Vulnerabilities in Block Cipher Implementations: Despite being a known risk for cache side-channel vulnerabilities, popular block cipher implementations like AES-256 from *OpenSSL* [13] and DES, 3DES, and Camellia from *mbedTLS* [19] continue to use large lookup tables for computational efficiency. These are the default choices in the absence of hardware acceleration. Examples for types

*Please cite the conference version of this paper [23] that was published in 61st Design Automation Conference (DAC).

Tool / Implementation	Obfuscation Technique	Automatic Hardening	Verified Leakage	No Hardware Extension	Partial Hardening
Bitsliced DES (FSE'97)	bitslicing	✗	✗	✓	✗
Bitsliced AES-GCM (CHES'09)	bitslicing	✗	✗	✗ (Intel SSE)	✗
Racoon (USENIX'15)	transactional + ORAM	✓	✗*	✗ (AVX)	✗
SC-Eliminator (ISSTA'18)	transformation + pre-loading	✓	✗*	✓	✗
RiCaSi (CANS'20)	circuit compilation	✓	✓	✓	✗
Constantine (CCS'21)	linerization	✓	✗*	✗ (AVX)	✗
HyCaMi	Secure LUTs	✓	✓	✓	✓

Table 1: Comparison to other Hardening Approaches (*performance optimizations prevent verification of hardened program)

of cache side-channel attacks include measuring runtime [3], learning which cache lines have been evicted by the target program [14], abusing access latency to shared memory buffers [35] and analyzing cache hits and misses from power traces [4]. Traditional mitigation involves constant-time programming techniques, such as bitslicing [5] which involves manually translating a block cipher specification to a circuit representation of logical bit operations. This manual approach has later been optimized for throughput using SIMD instructions for parallelism [17].

Static Cache Side-Channel Assessment: Qualitative tools like *CacheS* [31] and *CaType* [16] focus on the detection and localization of side-channel vulnerabilities. In this work we build on the family of tools *CacheAudit*, a quantitative approach that provides information theoretic upper bounds on the cache side-channel leakage of x86 binaries. The approach and initial implementation was published in [12] and later extended by the original authors [11] and authors of this paper [22, 24, 32]. Other quantitative tools include works on timing-attackers without cache on the level of C [15] and Java [20].

LUT-Based Circuits in Secure Multiparty Computation: In Secure Multiparty Computation (MPC), LUT-based protocols like FLUTE [7] and LUC [10] have enhanced efficiency by reducing cryptographic operations and circuit complexity. The tools for creating LUT-based circuits primarily originate from hardware synthesis targeting FPGAs. Yosys [33], an open-source framework, maps Verilog designs into circuits for various hardware targets, and has been employed in an MPC context [27]. To bridge the gap for programmers accustomed to imperative languages, high-level synthesis tools like XLS [1] from Google have been developed, which compile C/C++ code into Verilog, thus facilitating the design of MPC circuits in familiar programming languages.

This work: Following the exploration of LUTs in MPC as previously discussed, we now focus on the following question:

> *Are programs automatically hardened using LUTs more efficient in performance compared to those automatically hardened with Boolean circuits, while retaining the same cache side-channel security?*

To address this question, we introduce HyCaMi, an innovative approach that merges LUT synthesis with quantitative side-channel analysis. We proceed by elaborating on our exact contributions.

1.2 Our Contributions

In this work, we pivot the traditional understanding of LUTs in the realm of cybersecurity, transitioning from their conventional role as a vulnerability in side-channel attacks against block ciphers to a robust defensive mechanism. This novel application necessitates the development of new, optimized LUT-based circuit representations. Acknowledging the complexity and error-proneness of manual construction, we introduce an innovative automated toolchain. This toolchain adeptly transforms high-level function descriptions into efficient multi-input, multi-output LUT representations, leveraging repurposed hardware synthesis tools beyond their original purposes. Our approach, while producing binaries that may be slower compared to existing methods like Raccoon [28], SC-Eliminator [34], and Constantine [6], achieves superior security guarantees via static analysis of the hardened binaries. Compared to *RiCaSi* [22], the only other tool with comparable security properties on the binary, we achieve a speed that is up to 9.6× faster. Our main technical contributions are summarized as follows:

- **Encoding LUT-Based Circuits into C:** We introduce a cutting-edge method for encoding LUT-based circuits into C. This method strategically balances security and performance, offering either complete security against cache side-channels or significantly minimizing cache side-channel leakage, depending on the LUTs' size.
- **Development of HyCaMi Framework:** We develop HyCaMi, a comprehensive framework for the automatic hardening of block ciphers. HyCaMi synergizes LUT-based high-level synthesis with quantitative static side-channel analysis, culminating in a pipeline that automates the hardening of C/C++ source code. Our framework is open-sourced at https://encrypto.de/code/HyCaMi.
- **Extensive Evaluation Across Multiple Implementations** Our framework's efficacy is rigorously tested across four AES-256 implementations from *OpenSSL*, *mbedTLS*, *Nettle*, and *LibTomCrypt*, and implementations of DES, 3DES, and Camellia from *mbedTLS*. We demonstrate that HyCaMi produces binaries that are free of cache side channels and which are up to 9.6× faster than the hardened binaries of state-of-the-art work [22].
- **Exploring Security-Runtime Trade-Offs:** We explore, *for the first time*, the security-runtime trade-off induced by increasing size of LUTs. Applied to the same block ciphers, we show that in this configuration the binaries are up to 4.5× faster when compared to the fully secure variants, while for access-based attackers only having at most 18% of the leakage bound of the original program.

2 PRELIMINARIES

In this section we give a quick overview on the automatic quantification of cache side channels using program analysis. We calculate an upper bound of the capacity of a discrete memory-less channel $C : I \longrightarrow O$, where I is a finite set of secret inputs, O is a finite set of side-channel outputs and C models the behavior of the target program. Leakage is defined as the difficulty of guessing the secret input given the side-channel output. In other words, leakage is the difference in uncertainty of the attacker over the secret input before and after seeing the side-channel output. If this difference in uncertainty is given in terms of min-entropy (called min-entropy leakage), this value provides a measure on the reduction of guesses for a one-shot attacker in bits [29]. As also noted in [29], an upper bound of the min-entropy leakage can be calculated by counting the number of possible side-channel observations (i.e. elements of O). The family of tools *CacheAudit*, applies this approach to cache side channels on the x86 architecture. The tool overapproximates the set O by applying abstract interpretation [8] with an abstract model of the x86 architecture with cache. We categorize four distinct models of cache-side-channel adversaries, identified as *acc*, *accd*, *trace*, and *time*. Each adversary is defined in terms their set of possible side-channel observations O^a, where a represents one of the adversary models. They are:

- O^{acc}: The set of all cache states after termination of the victim program. In this model the attacker can deduce which memory blocks of the victim program are cached in a shared cache.
- O^{accd}: The set of all cache states after termination of the victim program. This attacker is similar to *acc*, but can only deduce how many memory blocks the victim program has loaded in each cache set of a shared cache.
- $O^{trace} \subseteq \{hit, miss, none\}^*$: The set of sequences (traces) of cache interactions. These include all instances of cache hits, misses, and cases of 'no access'. Such traces offer a detailed view of the cache behavior throughout a program's execution.
- $O^{time} \subseteq \mathbb{N}$: The set of possible running times as influenced by the caching behavior. These times are are calculated for a fixed duration for cache hits, misses, and non-memory accessing instructions.

These adversary models have been previously implemented for the *CacheAudit* family of tools. This pre-existing implementation allows us to integrate these models into our framework seamlessly, without necessitating any modifications to the existing code.

3 SECURE LUTS IN C

Whether a LUT with secret-depended accesses is a potential cache-side-channel vulnerability or not depends on the size of the LUT, how it is positioned in memory and how it is accessed by the program. In this section we present a novel technique for placing and accessing LUTs that is either fully side-channel secure, or minimizes leakage. The technique works for multi-input multi-output LUTs with at most 8 inputs and outputs. For this section we assume a target cache with 32 KiB size and a line size of 64 byte, the specification for a typical L1 data cache of Intel 8th Gen or AMD Zen 3 desktop chips. Only minor changes are required to adapt this technique to other caches of differrent size.

We distinguish two cases. If all LUTs have at most 6 inputs, we align them to 64 byte boundaries (i.e. the size of one cache line). If any LUT can have more than 6 inputs, we take two precautions to minimize leakage. First we place all LUTs at 32KiB boundaries (i.e. the size of the cache). Second, we rearrange the values in each table such that every access refers to the same cache set. For the cache described above, we place every value within 64 byte blocks that are located 4KiB apart (i.e. the number of cache sets multiplied by the cache line size). Listing 1 shows the source code used for an example 8-input 6-output LUT that was hardened.

Listing 1: Excerpt of generated C-code accessing tables for SecLUT-8.

```
static const uint8_t table_3[] __attribute__((aligned
    (32768))) = {0b00111111, ...};
...
uint32_t addr_3 = wire_0 | wire_2<<1 | wire_1<<2 | wire_3
    <<3 | wire_5<<4 | wire_7<<5 | wire_6<<6 | wire_4<<7;
addr_3 = ((addr_3 >> 6) * 4096) + (addr_3 % 64);
uint8_t tmp_3 = table_3[addr_3];
uint32_t wire_24 = tmp_3 & 0b1;
uint32_t wire_25 = (tmp_3 & 0b10) > 0;
uint32_t wire_26 = (tmp_3 & 0b100) > 0;
uint32_t wire_27 = (tmp_3 & 0b1000) > 0;
uint32_t wire_28 = (tmp_3 & 0b10000) > 0;
uint32_t wire_29 = (tmp_3 & 0b100000) > 0;
```

Following this method, if all LUTs have 6 inputs or less, the hardened binary is fully secure against cache side-channels by design. This is because every table takes $2^6 \cdot 8$ bits $= 64$ bytes in memory and thus fits into one cache line. Hence, even if the access location to the LUT is secret-dependent, no leakage occurs. LUTs with more than 6 inputs do not fit into one cache line. To minimize leakage we place them such that every access goes to the same cache set. For example, without hardening, a LUT with 8 inputs takes $2^8 \cdot 8$ bits $= 256$ bytes of memory and thus span four cache lines. After the hardening step, each 64 byte block is placed 4KiB apart such that all four cache lines fall into the same cache set. The result is that all LUT accesses are cached in the same cache set. Thus, if enough different LUTs are accessed (e.g. 8 for the LRU replacement strategy) the cache lines of prior LUT accesses are evicted, thereby hiding information from a cache side-channel attacker. For an access-based attacker without shared memory (i.e. attackers modeled by *accd*), this limits the amount of possible different observable states to 9, or at most $log_2(9) \approx 3.17$ bits of leakage. We can not make such predictions for the other three attacker models.

4 THE *HyCaMi* FRAMEWORK

The general workflow of HyCaMi is depicted in Figure 1. Given a C implementation of a block cipher that is possibly vulnerable to cache side-channel attacks, the source code is compiled to x86 machine code and statically analyzed with *CacheAudit v0.3* [12, 22]. If the leakage bound is acceptable, no hardening is required and the framework simply supplies these leakage guarantees on the binary. If, however, the leakage bound is too high, there are two paths. In both paths, the C source code is translated into an equivalent function following our synthesis flow. For the default case, LUTs with at most 6 inputs are chosen, which results in a fully secure program (cf. Section 3). We call this configuration SecLUT-6. Alternatively, the synthesis flow is applied using LUTs with at most 8 inputs. This results in a program that is potentially leaky with likely faster run times. We call this configuration SecLUT-8.

Both binaries are analyzed with *CacheAudit*. For the fully hardened version, this step statically verifies that the compiler did not introduce new unexpected leakage. In the default case, this verification passes. For the SecLUT-8 version, this step attests whether the

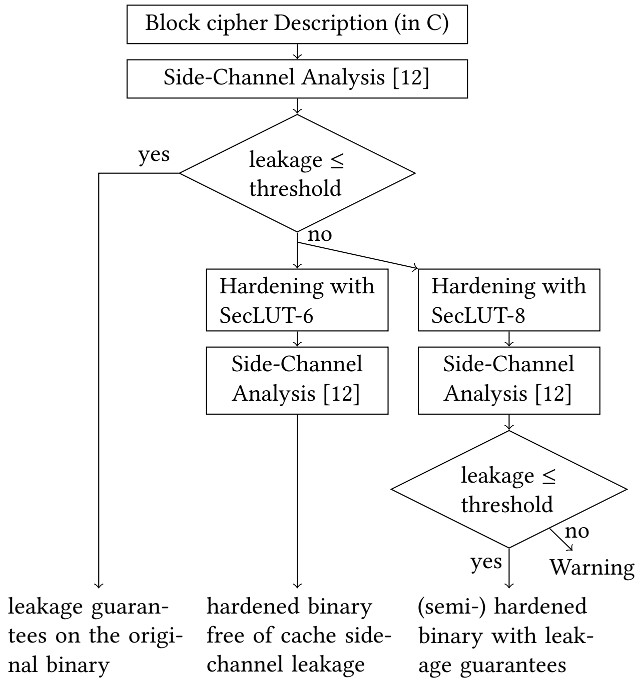

Figure 1: Workflow of HyCaMi

binary may be suited for the chosen scenario. If not, the framework still supplies this leaky variant together with a warning.

Synthesis: On a high level, *HyCaMi* translates a C implementation of a function into an equivalent C implementation based on secure LUT access by using open-source frameworks integrated into a custom compilation workflow. An overview of all processing steps is given in Figure 2. First, the function is manually extracted from the source C file. Depending on the implementation of the function, it may be necessary to modify the source code manually and replace C/C++ constructs not supported by XLS [1]. The functions studied in this paper required the conversion of unbounded `while` loops containing `break` statements to bounded `for` loops. Additionally, the implementations use pointer arithmetic which were replaced with array indexing operators. Given an implementation in C/C++ containing only supported constructs, XLS [1] translates the C/C++ code into an intermediate representation (IR) of a Boolean circuit along with additional metadata. This metadata describes the translation of the function signature in C to the circuit inputs and outputs in the IR. After applying optimizations, XLS translates the IR into Verilog code. The Verilog code generator is configured to output combinational Verilog circuits instead of pipelined circuits. Pipelined circuits achieve higher throughput by exploiting parallelism in hardware designs. The evaluation of the resulting LUT circuit happens in a single thread in software, thus the benefits of pipelining do not apply.

We use Yosys [33] to synthesize the Verilog code into Boolean circuits containing multi-input, single-output LUTs. In this step, Yosys also performs optimizations to reduce the circuit size. The synthesis result is provided in the BLIF format, which is then converted directly into the Secure Hardware Definition Language (SHDL) [21] format using a tool from the LUC framework [10]. Afterwards, another custom processing script from LUC merges multiple single-output LUTs into multi-output LUTs, such that the number of output wires per LUT is at most 8. This is due to the encoding of the LUT data which is discussed in detail in Section 3.

This circuit containing multi-input, multi-output LUTs is translated into C code using our novel LUT2C converter. The LUT2C translator requires information on the cache architecture of the target processor in order to layout the lookup table bitstring in a way which minimizes the information revealed in side-channels about the table index accessed. The resulting C source code is divided into three parts: The unwrapping of function parameters into individual wire values, the calculation of the results, and the wrapping of wire values back into C types. The unwrapping and wrapping code is generated using the metadata provided by XLS in the first step of the compilation pipeline. This makes it possible for the generated function to have the same function signature as the original implementation.

The calculation of the result consists of a series of operations for each LUT gate. First, the index into the lookup table data is calculated by concatenating the input wire value bits. The data in the corresponding index is then read from a byte array. Depending on the number of outputs, the bits contained in the byte are assigned to multiple wires. We describe the access in detail in Section 3. The generated code can then be compiled into a binary object file and linked with the original code with a C/C++ compiler.

5 PERFORMANCE EVALUATION

We evaluate the effectiveness of hardening and the overhead incurred by HyCaMi at the example of seven block cipher software implementations. First, we apply HyCaMi to DES, 3DES and Camellia from *mbedTLS* 2.16.5 [19]. All three implementations use LUTs to speed up computation. Such LUT-based implementations of DES, and Camellia are known to be vulnerable to cache side-channel attacks [30]. Second, we apply HyCaMi to AES-256 implementations from *OpenSSL* 1.1.1d [13], *mbedTLS* 2.16.5 [19], *Nettle* 3.5 [25] and *LibTomCrypt* 1.18.2 [18]. Despite implementing the same block cipher, the library authors utilized LUTs of different sizes, resulting in different cache-side-channel leakage characteristics [24]. For all block ciphers, we consider the implementation of the key schedule and the encryption algorithm. For the hardening, we have extracted all LUTs into individual functions and replaced all LUT accesses with calls to these functions. We then apply our new method to these extracted functions as described in Section 4.

Setup: For the evaluation we use an Intel i9-10900K, considering an attacker with access to a process running on the same system observing a shared L1 data cache. This cache has a size of 32 KiB, a line size of 64 byte and an associativity of 8.[1] We assume an LRU cache line replacement policy, since to the best of our knowledge, the replacement policy is not publicized for this CPU. We confirm the functional correctness of the hardened AES-256 binaries using test vectors from the NIST Cryptographic Algorithm Validation Program [2]. For the DES, 3DES and Camellia binaries we compare the output of the original program to the hardened variants for 10.000 random generated inputs.

[1]This information was obtained using the command-line tool *lstopo*.

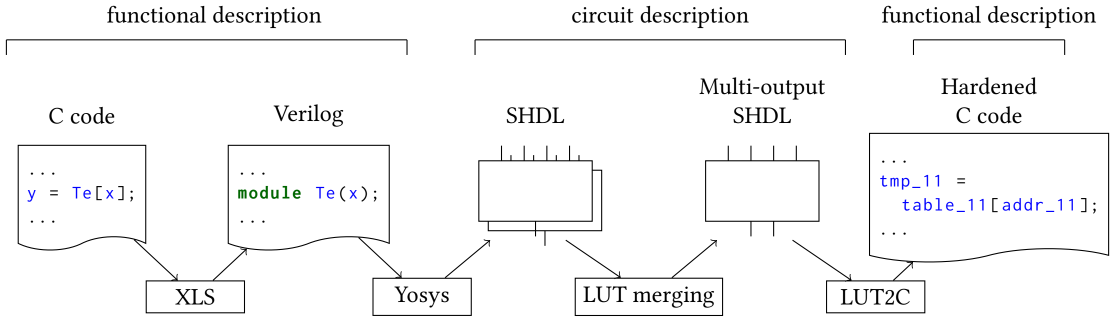

Figure 2: Synthesis flow

5.1 Evaluation of generated Circuits / LUTs

In Table 2, we give a brief overview of the distribution of different types of LUT in the generated circuits. A difference between DES/3DES and the other two block ciphers is immediately visible. AES and Camellia both use 8-input, 8-output S-boxes, whereas DES uses 6-input, 4-output S-boxes. In the AES implementations we analyzed, these S-boxes are not used directly, but combined with other operations into larger 8-input 32-output LUTs for increased performance. We thus conclude that these underlying differences in the structure lead to different characteristics in the LUT distribution.

We can also see how the number of possible inputs affects the distribution and overall number of gates in the resulting circuit. Due to allowing for more inputs to be gathered in one gate, the 8-input LUT circuits have consistently lower gate counts. Averaged over the four AES implementations tested, the 6-input circuits have ~ 14× more LUTs. The 6-input circuits for Camellia and DES/3DES are 8.4× and ~ 4.3× larger, respectively. This has an effect on the runtime performance, which is discussed in Section 5.2. Also noteworthy are the high numbers of 1-input, 6-output and 1-input, 7-output gates. As a 1-input LUT can only have 2 distinct outputs, further output wires are redundant. This issue occurs because the LUT merger treats individual LUTs as black boxes, making it unable to group output wires together based on their function. Removing this limitation is worth investigating in the future.

5.2 Runtime Overhead

Table 3 illustrates the runtime analysis of programs hardened with 6-input and 8-input LUTs, compared to their original versions. This analysis involved averaging the runtime over 1,000,000 executions of the key schedule and encryption functions for each covered implementation, further averaged across 20 repetitions to ensure consistency. The results confirm a significant increase in runtime overhead due to the hardening process, particularly impacting ciphers like AES-256 with larger original LUTs more than those with smaller LUTs, such as DES and 3DES. Notably, the data also reveals that the binaries hardened with the SecLUT-8 configuration exhibit enhanced performance, being 2.6× to 4.5× faster than those in the SecLUT-6 configuration, thereby indicating that larger LUTs in the hardening step contribute to improved runtime efficiency.

5.3 Leakage Assessment

To evaluate the cache side-channel leakage we use the static analysis tool *CacheAudit v0.3*. For both case studies we analyze wrapper programs that first set up library specific data structures and then call the key schedule and encryption function. The key and message are left uninitialized such that *CacheAudit* considers them as private input with unknown value. The wrapper programs are compiled with *gcc 9.3.0* using the parameters *-m32-fno-stack-protector* to obtain x86 binaries compatible with the tool. The analysis results are depicted in Table 4. As expected the results for the original programs reveal potential side-channel leakage across all considered block cipher implementations. The evaluation confirms for the examples the SecLUT-6 configuration is fully secure against cache side-channels, while programs hardened via the SecLUT-8 configuration are still potentially leaky but have reduced bounds when compared to the original program. Moreover, as predicted the leakage bounds for *accd* stayed below 3.17 bits. We therefore have shown that HyCaMi offers two types of hardening depending on developer needs. Either the framework generates a fully secure binary or a faster binary is leaky but still has better side-channel security properties than the original program.

	Library	Cipher	Inputs/Outputs				
			4/1	6/1	6/2	6/5	other
SecLUT-6	mbedTLS	DES	0	150	21	3	85
	mbedTLS	3DES	0	150	21	3	85
	mbedTLS	Camellia	60	0	16	20	72
	mbedTLS	AES-256	167	0	26	28	121
	OpenSSL	AES-256	152	0	22	25	100
	Nettle	AES-256	167	0	26	28	121
	LibTomCrypt	AES-256	170	0	24	29	120
			1/6	8/5	1/7	8/7	other
SecLUT-8	mbedTLS	DES	20	10	10	0	20
	mbedTLS	3DES	20	10	10	0	20
	mbedTLS	Camellia	0	0	8	8	4
	mbedTLS	AES-256	0	4	2	6	13
	OpenSSL	AES-256	0	4	0	4	12
	Nettle	AES-256	0	4	2	6	13
	LibTomCrypt	AES-256	0	4	0	4	17

Table 2: Distribution of LUT gate sizes in generated circuits

5.4 Comparison to Related Methods

We compare our new framework to *RiCaSi*, the only framework that generates hardened binaries that are statically verified to be secure against cache side-channels. For the comparison, we apply *RiCaSi*

Library	Cipher	Orig.	SecLUT-6	SecLUT-8
mbedTLS	DES	0.25 μs	6.0 μs/ 24.0×	2.3 μs/ 9.2×
mbedTLS	3DES	0.61 μs	17.8 μs/ 29.5×	6.9 μs/11.3×
mbedTLS	Camellia	0.32 μs	14.7 μs/ 46.0×	4.0 μs/12.6×
mbedTLS	AES-256	0.16 μs	26.8 μs/143.3×	5.9 μs/37.4×
OpenSSL	AES-256	0.16 μs	24.6 μs/156.3×	6.5 μs/40.9×
Nettle	AES-256	0.22 μs	22.8 μs/103.3×	5.7 μs/25.7×
LibTomCrypt	AES-256	0.89 μs	23.5 μs/ 26.5×	6.8 μs/ 7.6×

Table 3: Runtime Performance Evaluation on Intel i9-10900K

	Library	Cipher	Attacker Model			
			acc	*accd*	*trace*	*time*
Original	mbedTLS	DES	32.6	32.0	120.0	6.9
	mbedTLS	3DES	32.6	32.0	376.0	8.5
	mbedTLS	Camellia	17.8	16.0	236.0	7.9
	mbedTLS	AES-256	71.0	66.3	275.0	8.1
	OpenSSL	AES-256	68.1	64.5	274.0	8.1
	Nettle	AES-256	71.6	66.3	271.0	8.0
	LibTomCrypt	AES-256	161.3	99.3	274.0	8.1
SecLUT-6	mbedTLS	DES	0.0	0.0	0.0	0.0
	mbedTLS	3DES	0.0	0.0	0.0	0.0
	mbedTLS	Camellia	0.0	0.0	0.0	0.0
	mbedTLS	AES-256	0.0	0.0	0.0	0.0
	OpenSSL	AES-256	0.0	0.0	0.0	0.0
	Nettle	AES-256	0.0	0.0	0.0	0.0
	LibTomCrypt	AES-256	0.0	0.0	0.0	0.0
SecLUT-8	mbedTLS	DES	2.0	0.0	14.0	3.9
	mbedTLS	3DES	2.0	0.0	28.0	4.9
	mbedTLS	Camellia	4.0	0.0	58.0	5.9
	mbedTLS	AES-256	13.4	2.6	143.0	7.2
	OpenSSL	AES-256	9.3	1.0	0.0	0.0
	Nettle	AES-256	13.4	2.6	135.0	7.0
	LibTomCrypt	AES-256	9.7	2.0	88.0	6.5

Table 4: Cache Side-Channel Leakage Bounds in [bit] for a 32KiB, 8-way associative, cache with 64 byte cache lines and LRU policy

to the same block cipher implementations as HyCaMi. Table 5 shows the comparison of runtimes for the combined hardened key schedule and encryption implementation. Since *RiCaSi* only offers the option to completely harden a program, the comparison focuses on the fully secure SecLUT-6 configuration. Our comparison shows that HyCaMi is up to 9.6× faster than *RiCaSi*. Moreover, we can also see that the speedup is higher for Camellia, and the four AES-256 implementations. That is, our implementation is more effective for the four block ciphers with larger LUTs in the initial program.

6 CONCLUSION

This paper introduces HyCaMi, a pioneering framework designed for enhancing the security of block cipher implementations against cache side-channel attacks. This advancement leverages LUT-based high-level synthesis combined with quantitative side-channel analysis. Central to our methodology is a novel technique for translating LUT-based circuits into C code. This allows us to delve into the trade-offs between side-channel security and operational efficiency by varying the LUT sizes. Our implementation of HyCaMi yields two distinct outcomes: one is a 6-input LUT-based binary inherently immune to cache side-channel attacks, and the other is an 8-input LUT-based binary that maintains minimal leakage potential.

To assess the efficacy of our approach, we applied *HyCaMi* to seven varied block cipher implementations. The results of our evaluation indicate that binaries hardened with 6-input LUTs are completely secure against cache side-channel leakage. In contrast, those hardened with 8-input LUTs demonstrate considerably lower leakage bounds under specific cache side-channel attack models when compared to the original program. Furthermore, as delineated in Section 5.4, our analysis conclusively addresses our research question, establishing that LUT-hardened programs not only exhibit superior efficiency (with performance improvements up to 9.6×) compared to those hardened via Boolean circuits but also maintain equivalent levels of cache-side-channel security.

Library	Cipher	HyCaMi	*RiCaSi* [22]	Speedup
mbedTLS	DES	6.0 μs	7.0 μs	1.2×
mbedTLS	3DES	17.8 μs	26.8 μs	1.5×
mbedTLS	Camellia	14.7 μs	103.5 μs	7.0×
mbedTLS	AES-256	26.8 μs	216.8 μs	8.1×
OpenSSL	AES-256	24.6 μs	214.9 μs	8.7×
Nettle	AES-256	22.8 μs	217.9 μs	9.6×
LibTomCrypt	AES-256	23.5 μs	225.7 μs	9.6×

Table 5: Comparing the runtime of key schedule and encryption between ciphers hardened with HyCaMi (SecLUT-6) and RiCaSi [22]

ACKNOWLEDGMENTS

This project was funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the ERC under the EU's Horizon 2020 program (grant No. 850990 PSOTI).

REFERENCES

[1] 2020. XLS: Accelerated HW Synthesis. https://google.github.io/xls.
[2] Lawrence E Bassham III. 2002. The advanced encryption standard algorithm validation suite (AESAVS). *NIST Information Technology Laboratory* (2002).
[3] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report. University of Illinois at Chicago.
[4] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. 2005. AES power attack based on induced cache miss and countermeasure. In *ITCC*.
[5] Eli Biham. 1997. A fast new DES implementation in software. In *FSE*.
[6] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS*.
[7] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2023. FLUTE: Fast and Secure Lookup Table Evaluations. In *S&P*.
[8] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN*.
[9] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *S&P*.

[10] Yann Disser, Daniel Günther, Thomas Schneider, Maximilian Stillger, Arthur Wigandt, and Hossein Yalame. 2023. Breaking the Size Barrier: Universal Circuits meet Lookup Tables. In *ASIACRYPT*.
[11] Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *ACM PLDI*.
[12] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM TISSEC* 18, 1 (2015).
[13] OpenSSL Software Foundation. 2023. *OpenSSL (Version 1.0.1d)*.
[14] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *IEEE S&P*.
[15] Jonathan Heusser and Pasquale Malacaria. 2010. Quantifying Information Leaks in Software. In *ACM ACSAC*.
[16] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In *CCS*.
[17] Emilia Käsper and Peter Schwabe. 2009. Faster and timing-attack resistant AES-GCM. In *CHES*.
[18] libtom projects. 2018. *LibTomCrypt (Version 1.18.2)*.
[19] ARM Limited. 2023. *mbedTLS (Version 2.16.5)*. https://tls.mbed.org/download/start/mbedtls-2.16.5-apache.tgz
[20] Pasquale Malacaria, MHR. Khouzani, Corina S. Păsăreanu, Quoc-Sang Phan, and Kasper S. Luckow. 2018. Symbolic Side-Channel Analysis for Probabilistic Programs. In *IEEE CSF*.
[21] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. 2004. Fairplay-Secure Two-Party Computation System.. In *USENIX Security*.
[22] Heiko Mantel, Lukas Scheidel, Thomas Schneider, Alexandra Weber, Christian Weinert, and Tim Weißmantel. 2020. RiCaSi: Rigorous Cache Side Channel Mitigation via Circuit Compilation. In *CANS*.
[23] Heiko Mantel, Joachim Schmidt, Thomas Schneider, Maximilian Stillger, Tim Weißmantel, and Hossein Yalame. 2024. HyCaMi: High-Level Synthesis for Cache Side-Channel Mitigation. In *DAC*.
[24] Heiko Mantel, Alexandra Weber, and Boris Köpf. 2017. A Systematic Study of Cache Side Channels across AES Implementations. In *ESSOS*.
[25] Niels Möller. 2019. *Nettle (Version 3.5)*. https://ftp.gnu.org/gnu/nettle/nettle-3.5.tar.gz
[26] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*.
[27] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation. In *HOST*.
[28] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security*.
[29] Geoffrey Smith. 2009. On the foundations of quantitative information flow. In *FoSSaCS*.
[30] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *CHES*.
[31] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *USENIX Security*.
[32] Alexandra Weber, Oleg Nikiforov, Alexander Sauer, Johannes Schickel, Gernot Alber, Heiko Mantel, and Thomas Walther. 2021. Cache-Side-Channel Quantification and Mitigation for Quantum Cryptography. In *ESORICS*.
[33] Clifford Wolf. 2016. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.
[34] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks Using Program Repair. In *ACM ISSTA*.
[35] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.