# A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries

Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, Clémentine Maurice

# A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries

Antoine Geimer
Univ. Lille, CNRS, Inria
Univ. Rennes, CNRS, IRISA
Lille, France

Mathéo Vergnolle
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Frédéric Recoules
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Lesly-Ann Daniel
KU Leuven, imec-DistriNet
Leuven, Belgium

Sébastien Bardin
Université Paris-Saclay, CEA, List
Gif-sur-Yvettes, France

Clémentine Maurice
Univ. Lille, CNRS, Inria
Lille, France

## Abstract

To protect cryptographic implementations from side-channel vulnerabilities, developers must adopt constant-time programming practices. As these can be error-prone, many side-channel detection tools have been proposed. Despite this, such vulnerabilities are still manually found in cryptographic libraries. While a recent paper by Jancar et al. shows that developers rarely perform side-channel detection, it is unclear if existing detection tools could have found these vulnerabilities in the first place.

To answer this question we surveyed the literature to build a classification of 34 side-channel detection frameworks. The classification we offer compares multiple criteria, including the methods used, the scalability of the analysis or the threat model considered. We then built a unified common benchmark of representative cryptographic operations on a selection of 5 promising detection tools. This benchmark allows us to better compare the capabilities of each tool, and the scalability of their analysis. Additionally, we offer a classification of recently published side-channel vulnerabilities. We then test each of the selected tools on benchmarks reproducing a subset of these vulnerabilities as well as the context in which they appear. We find that existing tools can struggle to find vulnerabilities for a variety of reasons, mainly the lack of support for SIMD instructions, implicit flows, and internal secret generation. Based on our findings, we develop a set of recommendations for the research community and cryptographic library developers, with the goal to improve the effectiveness of side-channel detection tools.

**ACM Reference Format:**
Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *Proceedings of 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX


*CCS '23, November 26–30, 2023, Copenhagen, Denmark*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Implementing cryptographic algorithms is an arduous task. Beyond functional correctness, the developers must also ensure that their code does not leak potentially secret information through side channels. Since Paul Kocher's seminal work [82], the research community has combed through software and hardware to find vectors allowing for side-channel attacks, from execution time to electromagnetic emissions. The unifying principle behind this class of attacks is that they do not exploit the algorithm *specification* but rather *physical characteristics* of its execution. Among the aforementioned attack vectors, the processor microarchitecture is of particular interest, as it is a shared resource between multiple programs. By observing the target execution through microarchitectural components (e.g., cache [88, 139], branch predictor [6, 57], DRAM [98], CPU ports [9]), an attacker can deduce secret information beyond what is normally possible with classical cryptanalysis. Side-channel primitives using these components allow an attacker to reconstruct secret-dependent control flow and table look-ups. This problem is exacerbated in multi-core processors and VM environments, where execution can be shared concurrently by multiple actors, and in trusted execution environments, where the privileged control of untrusted operating systems can be leveraged to perform controlled-channel attacks [137].

Consequently, multiple countermeasures to timing and microarchitectural attacks have been developed in the literature. System-level approaches like StealthMem [79] modify the OS' behavior between context switches to minimize information sharing, while language-level approaches like type systems can be used to enforce proper information flow in the source code [12]. Hardware-based approaches also enable securing components by design [102, 143]. However, these approaches are hardly practical as they rely on either large source code rewrites, or require system or hardware modifications. Manually writing a program such that it is free of such microarchitectural leakage is thus, by far, the most commonly employed countermeasure in cryptographic libraries [1, 26]. In particular, the *constant-time programming* discipline [22, 26] (CT) consists in writing a program such that its control flow, memory accesses and operands of variable-time instructions do not depend on secret data, and is considered the de-facto standard to protect against (non-transient) timing and microarchitectural attacks. Constant-time programming is an arduous task as its recommendations go against usual programming practices and requires knowledge of the literature on side channels. Moreover, the programmer must

be mindful of compiler optimizations not preserving constant-time code [52, 115] and libraries not following these practices at all [78].

**Problem.** In the past decade, the research community has investigated *automated* ways of checking whether a program is leakage-free or not. Many different approaches have been proposed [20, 51, 66, 85, 106, 132, 133], both static and dynamic, but despite this abundance of tools, side-channel vulnerabilities are still found regularly in cryptographic libraries [89]. Two factors could explain this paradox: either these tools were not able to find such vulnerabilities, or the developers did not use them. A recent survey by Jancar et al. [73] investigated the opinions of cryptographic libraries developers regarding side-channel detection frameworks. They found that while many were willing to include side channels in their threat models, very few actually used tools. While this provides answers for the second factor, the first one remains unexplored.

**Goal.** Our paper investigates this question by providing a thorough state of the art on side-channel detection frameworks and recent side-channel vulnerabilities. In particular, we restrict ourselves to vulnerabilities found in cryptographic libraries spanning five years (2017-2022). We only consider vulnerabilities exploited through passive microarchitectural side-channel attacks. Physical side channels [43, 63, 83, 105], active attacks (e.g., Rowhammer [80]) or transient execution attacks [44, 45, 103, 124] are out-of-scope.

We tackle the following research questions:

**RQ1** How to compare these frameworks, as their respective publications offer differing evaluation?

**RQ2** Could an existing framework have detected these vulnerabilities found manually?

**RQ3** What features might be missing from existing frameworks to find these vulnerabilities?

**Contributions.** The contributions can be summarized as follows:

(1) We present a qualitative classification of the state-of-the-art tools for side-channel vulnerability detection. We classify 34 frameworks depending on multiple parameters such as the methods used, the type of outputs given by the analysis, or the type of programs analyzed (Section 4);
(2) We compare a subset of these detection frameworks on a unified benchmark, comprised of representative cryptographic operations from 3 libraries, totaling 25 primitives. We found that asymmetric cryptographic primitives are still a challenge for most detection tools. This benchmark aims at ensuring a fair comparison between frameworks and help develop future efforts (Section 7);
(3) We offer a classification of recently published side-channel vulnerabilities in cryptographic libraries, offering new insights into where to find potential vulnerabilities (Section 5);
(4) We verify whether 4 of these vulnerabilities could have been detected with the aforementioned frameworks. We conclude that support for SIMD instructions, implicit flows, and internal secret generation is crucial for effective side-channel detection tools (Section 8).

*Our whole evaluation framework is available for the community.*[1]

[1] https://github.com/ageimer/sok-detection/

# 2 Preliminaries

## 2.1 Scope

Language-based approaches for high-assurance cryptography—in particular used in the EverCrypt library [100, 144], FaCT [46], and Jasmin [12, 14]—provide provably functionally correct and constant-time implementations of cryptographic primitives and libraries, but require a complete code rewrite in a dedicated language. In this paper, we consider out-of-scope such language-based approaches and focus on tools that are applicable to off-the-shelf libraries.

Program transformations and repair tools have been proposed to transform insecure programs into (variants of) constant-time programs [7, 31, 35, 47, 84, 93, 104, 116, 135]. Because side-channel *detection* is not the main focus of these works, we only include them when they propose a novel detection phase.

Power side-channel attacks [83] exploit data-dependent differences in the power consumption of a CPU to extract secrets. These attacks usually assume a stronger attacker model than timing microarchitectural attacks, typically requiring physical access to a device. The tools we consider in this survey explicitly restrict to the latter attacker model, hence they cannot find vulnerabilities related to power-based attacks.

Interestingly, recent *frequency throttling* side-channel attacks [87, 129] exploit the fact that power consumption can also influence the execution time of a program via dynamic voltage and frequency scaling features of CPUs. These attacks effectively bring power side-channels to the (micro-)architecture, making them exploitable without physical access, and blur the boundary between these two attacker models. We consider these attacks out-of-scope as they also affect constant-time programs and require different mitigations, traditionally used to protect cryptographic code against power-side channel attacks, such as masking [65, 92] or blinding [75].

Finally, we focus on side-channel detection tools for *cryptographic code* and exclude tools targeting other libraries [128, 142].

## 2.2 Related Work

The closest related work is a recent survey by Jancar et al. [59, 73], which analyzes how cryptographic developers ensure that their code is secure against microarchitectural attacks. In particular, this survey classifies existing tools for microarchitectural side-channel analysis, and identifies: *(1)* whether developers are aware of these tools or have experience with them, *(2)* what kind of tools developers are willing to use (specifically given the trade-off between strong guarantees and usability), *(3)* what are the common shortcomings hindering the adoption of these tools. In contrast, we focus on technical differences between these tools in order to compare their actual capabilities. We also select a subset of promising tools and experimentally assess their scalability and ability to detect recent vulnerabilities from cryptographic libraries.

Yuan et al. [142] compared 11 side-channels detection tools on 8 criteria, including some overlapping with ours. However, our classification aims at being more general and systematic by considering more general criteria and a broader set of tools.

Lou et al. presented a survey [89] characterizing hardware and software vectors in microarchitectural side-channel attacks. While

their work is more exhaustive in terms of characterizing side-channel vulnerabilities in cryptographic libraries, our work is focused on vulnerabilities found in a recent period and its purpose is to understand why they were not found automatically.

Barbosa et al. [21] proposed a systematization of the literature and a taxonomy of tools for computer-aided cryptography. Their survey encompasses formal, machine-checkable approaches for design-level security, functional correctness, and side-channel security. In contrast, we focus on side-channel security but consider a broader set of tools, not restricted to provably secure approaches.

Complementary to our work, Ge et al. [62] proposed a taxonomy of timing microarchitectural attacks and defenses. They give an overview of microarchitectural components that are susceptible to side-channel attacks and classify attacks according to the degree of hardware sharing and concurrency involved. Finally, they survey existing hardware and software countermeasures against these attacks. Similarly, Jakub Szefer [118] proposed a survey of microarchitectural channels, attacks and defenses with a stronger focus on microarchitectural features enabling covert and side channels.

Buhan et al. [39] presented a taxonomy of tools for protecting against physical side-channel attacks such as power consumption or electromagnetic emanations. Their survey encompasses tools for physical side-channel leakage detection, verification, and mitigation for pre- or post-silicon development stage. Conversely, our work focuses on (non-overlapping) tools for timing and microarchitectural side-channels detection in software.

## 3 Background

This section introduces the background to understand side-channel vulnerabilities, in particular the hardware and software vectors.

### 3.1 Microarchitectural side channels

Microarchitectural attacks use shared microarchitectural components to deduce secret information from a program executing on the same physical core, CPU, or socket. By probing a component state, the attacker can observe the changes the victim's execution induced on that state, thus gaining information on the victim. The type and quantity of information gained depend on the component and the way it was probed, e.g., branch predictors can expose the direction of branches taken by a victim [5], port contention allows an attacker to deduce which instructions the victim executes [9], TLB attacks leak the victim's memory accesses at page-level resolution [67]. By far the most widely used component in attacks remains the cache. Multiple techniques have been developed to attack caches, to retrieve both the victim's control-flow and memory accesses, such as Prime+Probe [88, 96] and Flush+Reload [139]. Some of these attacks can also be run remotely [37, 38]. They can be especially powerful in the context of trusted execution environment, such as Intel-SGX, where the untrusted operating system can be leveraged to perform controlled-channel attacks, enabling high-resolution and low-noise side channels [40, 123, 137]. A complete overview of microarchitectural side-channel attacks can be found in [62, 118].

### 3.2 Side-channel vulnerabilities

The hardware vector represents only one aspect of side channels. To be exploitable, software must leak sensitive data in some way. Memory accesses and control flow that depend on the secret are the most commonly identified sources of leakage in cryptographic software. We detail common side-channel pitfalls encountered in cryptographc implementations. An exhaustive look at vulnerabilities in cryptographic libraries is presented in [89].

**Control-flow.** One example of secret-dependent control flow is the *square-and-multiply* implementation of modular exponentiation used in RSA, where the operation sequence depends on the bit value of the secret exponent, leading to timing attacks [82]. Scalar multiplication, the analogous operation for elliptic curve cryptography, suffers from a similar problem in *double-and-add* implementations. Variants of these implementations such as the *sliding window* and *wNAF multiplication* algorithms are preferred for their performance, but do not alleviate this vulnerability [24, 96]. To mitigate control-flow-based side-channel attacks, developers have resorted to either balancing [8] or linearizing [93] (*i.e.*, eliminating) secret-dependent control-flow. The former solution, employed for instance in the Montgomery ladder algorithm [76] for scalar multiplication, is particularly challenging to get right as it remains vulnerable to attacks exploiting port-contention, branch predictors, and (instruction and data) cache attacks [138]. The latter is not a fool-proof solution either as it remains vulnerable to attacks exploiting secret-dependent memory accesses [67]. Modular inversion is another common operation that can be a source of side-channel leakage, in particular through the control flow of greatest common divisor (GCD) algorithms, such as the binary extended Euclidean algorithm (BEEA) [5]. Secret-dependent branches are also at the heart of padding oracles used to mount Bleichenbacher's attack [29] or Lucky 13 [11].

**Memory access.** The original AES proposal describes how the cipher can be efficiently implemented using pre-computed tables with an access offset depending on the secret key, making it particularly vulnerable to cache attacks [25]. The aforementioned *sliding window* and *wNAF multiplication* algorithm also employs tables to store pre-computed values to speed up computations which, when accessed with a value derived from the secret, induce leakage through memory accesses [36, 88].

**Operand values.** Secret-dependent operand values can also be a source of leakage if they influence the program running time. Early termination in integer multiplication on ARM has been shown to induce a timing leak [69], with similar problem being found in x86 floating-point instructions [16].

## 4 Classifying side-channel vulnerability detection frameworks

As the concept of microarchitectural attacks has gathered attention only relatively recently, all the approaches we present here are less than a decade old. We structure this section in two large categories, static and dynamic, that represents a split in approaches but also in research communities: the first closer to program verification and the second to bug-finding. However, we strive to compare these approaches using broader parameters. Table 1 gives a comparison of the 34 detection frameworks we consider in this survey.

**Criteria.** In addition to comparing frameworks by their type (static or dynamic), we give a description of the precise method employed. While our efforts are independent from [59], there is a significant

**Table 1: Comparison table of vulnerability detection frameworks.**

| Ref | Year | **Tool** | Type | Methods | Scal. | Policy | Sound | Input | L | W | E | B | Available |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| [85] | 2010 | ct-grind | Dynamic | Tainting | ● | **CT** | ◐ | **Binary** | ✓ | | | | ✓ |
| [15] | 2013 | Almeida et al. | Static | Deductive verification | ○ | **CT** | ● | C source | | | | | |
| [55] | 2013 | CacheAudit | Static | Abstract interpretation | ○ | CO | ◐ | **Binary** | | | ✓ | | ✓ |
| [22] | 2014 | VirtualCert | Static | Type system | ○ | **CT** | ● | C source | | | ✓ | | ✓ |
| [70] | 2015 | Cache Templates | Dynamic | Statistical tests | ○ | CO | ○ | **Binary** | ✓ | | | | ✓ |
| [13] | 2016 | ct-verif | Static | Logical verification | ◐ | **CT** | ● | LLVM | | | | | ✓ |
| [107] | 2016 | FlowTracker | Static | Type system | ◐ | **CT** | ● | LLVM | ✓ | | | | ✓ |
| [56] | 2017 | CacheAudit2 | Static | Abstract interpretation | ○ | **CT** | ● | **Binary** | | | ✓ | | |
| [28] | 2017 | Blazy et al. | Static | Abstract interpretation | ◐ | **CT** | ● | C source | | | | | |
| [17] | 2017 | Blazer | Static | Decomposition | ◐ | CR | ● | Java | | ✓ | | | |
| [48] | 2017 | Themis | Static | Logical verification | ◐ | CR | ● | Java | ✓ | ✓ | | | |
| [127] | 2017 | CacheD | Dynamic | DSE | ◐ | CO | ○ | **Binary** | ✓ | ✓ | | | |
| [136] | 2017 | STACCO | Dynamic | Trace diff | ◐ | CR | ○ | **Binary** | ✓ | | | | ✓ |
| [106] | 2017 | dudect | Dynamic | Statistical tests | ◐ | CC | ○ | **Binary** | | | | | ✓ |
| [117] | 2018 | CANAL | Static | SE | ○ | CO | ◐ | LLVM | | ✓ | | | ✓ |
| [47] | 2018 | CacheFix | Static | SE | ◐ | CO | ◐ | C | ✓ | ✓ | | | ✓ |
| [34] | 2018 | CoCo-Channel | Static | SE, tainting | ● | CR | ◐ | Java | | ✓ | | | |
| [19] | 2018 | SideTrail | Static | Logical verification | ○ | CR | ● | LLVM | ✓ | ✓ | ✓ | | ✓ |
| [114] | 2018 | Shin et al. | Dynamic | Statistical tests | ◐ | CO | ○ | **Binary** | ✓ | | | | |
| [132] | 2018 | DATA | Dynamic | Statistical tests | ◐ | **CT** | ○ | **Binary** | ✓ | | | ✓ | ✓ |
| [133] | 2018 | MicroWalk | Dynamic | MIA | ● | **CT** | ○ | **Binary** | ✓ | | ✓ | | ✓ |
| [110] | 2019 | STAnalyzer | Static | Abstract interpretation | ● | **CT** | ● | C | ✓ | | | | ✓ |
| [95] | 2019 | DifFuzz | Dynamic | Fuzzing | ◐ | CR | ○ | Java | | ✓ | | | ✓ |
| [126] | 2019 | CacheS | Static | Abstract interpretation, SE | ● | **CT** | ○ | **Binary** | ✓ | ✓ | | | |
| [35] | 2019 | CaSym | Static | SE | ◐ | CO | ● | LLVM | ✓ | ✓ | | | |
| [54] | 2020 | Pitchfork | Static | SE, tainting | ● | **CT** | ◐ | LLVM | ✓ | ✓ | | | ✓ |
| [66] | 2020 | ABSynthe | Dynamic | Genetic algorithm, RNN | ◐ | CR | ○ | C source | ✓ | | | | ✓ |
| [72] | 2020 | ct-fuzz | Dynamic | Fuzzing | ◐ | **CT** | ○ | **Binary** | ✓ | ✓ | | | ✓ |
| [51] | 2020 | Binsec/Rel | Static | SE | ● | **CT** | ◐ | **Binary** | ✓ | ✓ | | | ✓ |
| [20] | 2021 | Abacus | Dynamic | DSE | ● | **CT** | ◐ | **Binary** | ✓ | | ✓ | | ✓ |
| [74] | 2022 | CaType | Dynamic | Type system | ◐ | CO | ● | **Binary** | ✓ | | | ✓ | |
| [134] | 2022 | MicroWalk-CI | Dynamic | MIA | ● | **CT** | ○ | **Binary**, JS | ✓ | | ✓ | | ✓ |
| [140] | 2022 | ENCIDER | Static | SE | ● | **CT** | ◐ | LLVM | ✓ | ✓ | | | ✓ |
| [141] | 2023 | CacheQL | Dynamic | MIA, NN | ● | **CT** | ○ | **Binary** | ✓ | | ✓ | ✓ | ✓† |

(D)SE: (Dynamic) Symbolic Execution, MIA: Mutual Information Analysis, (R)NN: (Recurrent) Neural Network, E: estimation of the number of bits leaked, L: origin of the leakage, W: witness triggering the vulnerability, B: support for blinding. † Not available at the time of writing.

overlap between the two classifications. Yet the present one goes into more details, being enriched with additional criteria. Similarly to [59, 73], we detail the type of programs supported by the analysis (e.g., C, Java, binary code) and whether the approach is supported by a soundness claim. While analyzing the source code or LLVM is more practical as the program semantics is more easily extracted than in the binary, doing so poses the risk of missing vulnerabilities introduced by the compiler [52, 78, 115]. A full soundness claim (●) offers a formal guarantee that the analysis will not, in principle, accept insecure programs and so should not yield any false negatives. In practice, this may not be the case because of vulnerabilities outside the analysis' threat model, or because of bugs in the detection tool. Conversely, partial soundness (◐) claims cover tools that are sound under some restrictions (e.g., partial exploration of loops). In addition, our classification reports the policy enforced by these tools: (CT) *constant-time* proscribes secret-dependent control flow, memory accesses, and operands of variable time instructions; (CO) *cache-obliviousness* requires the sequence of cache hits and misses w.r.t. a cache model to be independent from the secret input; (CR) *constant-ressource* requires the execution cost (determined by a fixed instruction cost), to be independent from the secret input; (CC) *constant clock cycles* requires the number of clock cycles to be independent from the secret input. Note that CT [13, 22] is strictly more conservative than CO and CR, and is the only policy that is secure w.r.t. the attacker scope considered in this paper.

We also detail the type of outputs given by the analysis: estimation on the number of bits leaked (E), origin of the leakage (L), and whether a witness triggering the vulnerability is given (W). A tool reporting neither these three only reports the presence or absence of vulnerabilities on the target. We additionally report whether the tool supports blinding [75] (B), a defense that introduces additional randomness to computations to hinder inference of secret values. Finally, we offer a rough estimation of the tool scalability.

**Limitation.** Unfortunately there is no clear metric to quantify scalability without testing each tool. One possibility could be the number of instructions a tool can process per second, however not all publications make this information available and the concept of *instruction* can vary from one approach to another (LoC, IR or assembly instructions, counting unrolled loops or not, etc.). Furthermore, such metric would only be valid for one set of benchmarks on the same machine, whereas publications generally include different benchmarks on a variety of setup. We provide here a rough estimation, based on the claims made in the tools' publications. We differentiate tools able to analyze in a reasonable time: complex asymmetric primitives (●), symmetric primitives (◐), and those struggling to scale even for these (○).

## 4.1 Static analysis

Static analysis approaches attempt to derive security properties from the program without actually *executing* it, extracting formally defined guarantees on all possible executions through binary or source code analysis. As a formal exploration of every reachable state is unfeasible, program behavior is often approximated, making them prone to false positives. Static approaches were the first to be considered, as side-channel security is closely related to information flow policies [53].

*4.1.1 Logical reduction* Non-interference is a *2-safety property* stating that two executions with equivalent public inputs and potentially different secret inputs must result in equivalent public outputs. This definition covers side channels by considering *resource usage* (e.g., address trace) as a public output. Approaches based on logical reduction to *1-safety* transform the program so that verifying its side-channel security amounts to proving the safety of the transformed program.

Self-composition [23] interleaves two executions of a program $P$ with different sets of secret variables in a single self-composed program $P; P'$. Solvers can then be used to verify the non-interference property. This approach was used by Bacelar Almeida et al. [15] to manually verify limited examples, relying on a large amount of code annotations. `ct-verif` instead runs the two copies in lockstep, while checking their assertion-safety [13]. It is able to verify LLVM programs, leveraging the BOOGIE verifier. Sidetrail [19] reuses this to verify that secret dependent branches are balanced (assuming a fixed instruction cost and excluding memory access patterns), providing a counter-example when this verification fails.

However, such approaches suffer from an explosion in the size of the program state space. Blazer [17] verifies timing-channel security on Java programs by instead decomposing the execution space into a partition on secret-independent branches. Proving 2-safety is thus reduced to verifying 1-safety on each trace in the partition improving scalability at the cost of precision. Themis [48] uses static taint analysis to automatically annotate secret-dependent Java code with Hoare logic formulas as pre- and post-conditions. An SMT solver then verifies that the post-condition implies execution time differences remain bounded by given constant. Both tools provide a witness triggering the vulnerability otherwise.

*4.1.2 Type systems* Approaches based on verifying type safety of a program differ from language-level countermeasures [12, 30], as the developer only needs to type the secret values with annotations instead of rewriting the program. The type system then propagates this throughout the program, similarly to static taint analysis. Type systems were considered relatively early to verify non-interference properties [7] and offer good scalability but their imprecision makes them difficult to use in practice.

VIRTUALCERT [22] analyzes a modified CompCert IR where each instruction makes its successors explicit. The authors define semantics for that representation, building the type system on top of it. An alias analysis giving a sound over-approximated prediction of targeted memory address is needed to handle pointer arithmetic. While this approach is more suited to a strict *verification* task, it can also provide a leakage estimate.

FlowTracker [107] introduces a novel algorithm to efficiently compute implicit information flows in a program, and uses it to apply a type system verifying constant-time.

*4.1.3 Abstract interpretation* As a program semantics is generally too complex to formally verify non-trivial properties, abstract interpretation [50] over-approximates its set of reachable states, so that if the approximation is safe, then the program is safe.

CacheAudit [55] performs a binary-level analysis, quantifying the amount of leakage depending on the cache policy by finding the size of the range of a side-channel function. This side-channel function is computed through abstract interpretation, and the size of its range determined using counting techniques. It was later extended to support dynamic memory and threat models allowing byte-level observations [56] and more x86 instructions [91].

Blazy et al. [28] focus on the source code instead of the binary. Their tool is integrated into the formally-verified Verasco static analyzer, and uses the CompCert compiler. The analysis is structured around a tainting semantics that propagates secret information throughout the program.

STAnalyzer [110] uses data-flow analysis to report secret-dependent branches and memory accesses.

CacheS [126] uses an hybrid approach between abstract interpretation and symbolic execution. The abstract domain keeps track of program secrets—with a precise symbolic representation for values in order to confirm leakage—but keeps only a coarse-grain representation of non-secret values. To improve scalability, CacheS implements a lightweight but unsound memory model.

*4.1.4 Symbolic execution* Symbolic execution [81] (SE) denotes approaches that verify properties of a program by executing it with symbolic inputs instead of concrete ones. Explored execution paths are associated with a logical formula: the conjunction of conditionals leading to that path. A memory model maps encountered variables onto symbolic expressions derived from the symbolic inputs and the concrete constants. A solver is then used to check whether a set of concrete values satisfies the generated formulas. Recent advances in SMT solvers have made symbolic execution a practical tool for program analysis [42].

CoCo-Channel [34] identifies secret-dependent conditions using taint-analysis, constructs symbolic cost expressions for each path of the program uses SE and reports paths that exhibit secret-dependent timing behavior. Their cost model assigns a fixed cost per instruction, excluding secret-dependent memory accesses.

Several works use symbolic execution to derive a symbolic cache model and check that cache behavior does not depend on secrets. CANAL [117] models cache behaviors of programs directly in the LLVM intermediate representation by inserting auxiliary variable and instructions. It then uses KLEE [41] to analyze the program and check that the number of hits does not depend on secrets. Similarly, CacheFix [47] uses SE to derive a symbolic cache model supporting multiple cache policies. In case of a violation, CacheFix can synthesize a fix by injecting cache hits/misses in the program. CaSym [35] follows the same methodology and, to improve scalability, includes simplifications of the symbolic state and loop transformations, which are sound but might introduce false positives.

SE suffers from scalability issues when applied to 2-safety properties like constant-time verification. Daniel et al. [51] adapt its formalism to binary analysis, introducing optimizations to maximize information shared between two executions following a same path. Their framework Binsec/Rel offers a binary-level CT analysis, performing a bounded exploration of reachable states and giving counterexamples for the identified vulnerabilities.

Pitchfork [54] combines SE and dynamic taint tracking. It soundly propagates secret taints along all executions paths, reporting tainted branch conditions or memory addresses. Interestingly, Pitchfork can analyze *protocol-level code* by abstracting away primitives' implementations using function hooks, and analyzing them separately.

ENCIDER [140] combines symbolic execution with taint analysis to reduce the number of solver calls. It also enables to specify information-flow function summaries to reduce path explosion.

## 4.2 Dynamic analysis

Dynamic analysis groups approaches that derive security guarantees from execution traces of a target program. Some form of dynamic binary instrumentation (DBI) is often used to execute the program and gather events of interest, such as memory accesses or jumps. Dynamic approaches differ in the events collected, and how traces are processed. They can be grouped depending on whether they reason on a single trace, or compare multiples traces together.

### 4.2.1 *Trace comparison approaches*

**Statistical tests.** Statistical tests can be used to check if different secrets induce statistically significant differences in recorded traces. Cache Template [70] monitors cache activity to detect lines associated with a target event, then finds lines correlated with the event using a similarity measure. A first pass using page-level observations instead of lines can be used to improve scalability [112]. Shin et al. [114] use K-means clustering to produce two groups of traces for each line. The confidence in the partition indicates which line is likely to be secret-dependent. DATA [132] employs a Kuiper test then a Randomized Dependence Coefficient test to infer linear and non-linear relationships between traces and secrets. This was later extended to support cryptographic nonces as secrets [130].

Mutual information (MI) can be used to quantify the information shared between secret values and recorded traces, with a non-zero MI score giving a leakage estimation. MicroWalk [133] computes MI scores between input sets and hashed traces, with leakage location pinpointed using finer-grained instruction-level MI scores. MicroWalk-CI [134] optimizes this process by transforming the traces in call trees, and adds support for JavaScript and easy integration in CI, following recommendations from [73]. CacheQL [141] reformulates MI into conditional probabilities, estimated with neural networks. Leakage location is estimated by recasting the problem into a cooperative game solved using Shapley values. Contrary to other tools [20, 133], CacheQL does not assume uniform distribution of the secret, nor deterministic executions traces.

STACCO [136] targets control-flow vulnerabilities specifically in TLS libraries running on SGX, focusing on oracles attacks [11, 29]. Traces recorded under different TLS packets are represented as sequences of basic blocks and compared using a diff tool.

Instead of recording traces, dudect [106] records overall clock cycles and compares their distribution with secret inputs divided in two classes (fix-vs-random). While this approach is simple and lightweight, it gives certainty that an implementation is secure *up to* a number of measurements. Contrary to other tools relying on an explicit leakage model, dudect directly monitors timings. Hence, vulnerabilities to other microarchitectural attacks like Hertzbleed might (in theory) be detected by dudect.

**Fuzzing.** Fuzzing techniques can be used to find inputs maximizing coverage and side-channel leakage. DifFuzz [95] combines fuzzing with self-composition to find side-channels based on instruction count, memory usage and response size in Java programs. ct-fuzz [72] extends this method to binary executables and cache leakage.

### 4.2.2 *Single trace*

Other approaches use only one trace to perform the analysis, sacrificing coverage for scalability. ctgrind [85] repurposes the dynamic taint analysis of Valgrind to check CT by declaring secrets as undefined memory. This solution is easy to deploy and reuses familiar tools, but remains imprecise.

ABSynthe [66] identifies secret-dependent branches using dynamic taint analysis. It employs a genetic algorithm to build a sequence of instructions based on interference maps evaluating contention created by each x86 instructions.

More precise approaches use SE to *replay* the trace with the secret as a symbolic value and check for CT violation. CacheD [127] applies this approach to memory accesses. Abacus [20] extends it to control-flow vulnerabilities, picking random values to check satisfiability instead of using a SMT solver. It also includes leakage estimation through Monte Carlo simulation.

Finally, CaType [74] uses refinement types (*i.e.*, types carrying a predicate restricting their possible values) on a trace to track constant bit values and improve precision. CaType also supports implementations that use blinding.

## 4.3 Insights

Despite the relative youth of the field, a wide variety of approaches have been proposed. While initially more static approaches were proposed, dynamic ones soon followed after 2017. This might represent a shift in research communities, from a focus on verification to bug-finding and, critically, scalability. Indeed, dynamic approaches typically scale better than static ones. Yet, this advantage mainly applies to single-trace analysis approaches. For trace comparison-based approaches, the scalability gain is less obvious as recording multiple traces can be time-consuming, particularly for statistical

approaches requiring a large number of traces, or for slower algorithms (e.g., RSA). Single trace analyses however suffer from a critical lack of coverage, which could be alleviated through methods like fuzzing. SE has become a popular approach for both static and dynamic methods, as recent advances in SMT solvers make it practical for side-channel detection.

Both the static and dynamic communities could benefit from integrating approaches from one another. For example we find that Abacus' optimization of trying random values to satisfy SMT formulas would pair well with Binsec/Rel's optimizations sparing UNSAT formulas.

## 5 Classifying side-channel vulnerabilities

We give here a (non-exhaustive) overview of microarchitectural side-channel vulnerabilities which were subject to publications in security and cryptography conferences in the past five years, many of which were still found manually by researchers. Interestingly, most of these vulnerabilities are new manifestations of already-known vulnerabilities (Section 5.1) and only few of them actually target new primitives or functionalities (Section 5.2).

### 5.1 Known vulnerabilities

Known vulnerabilities can resurface for two reasons: when known-vulnerable functions are used in *new contexts*, or in *new libraries*. In the first case, developers keep vulnerable functions in the code-base for performance reasons, carefully avoiding using them when manipulating secret data. This practice leaves the door open to new vulnerabilities in which these known-vulnerable functions (e.g., *square-and-multiply*) are used in a new context (e.g., key generation). In the second case, the lack of developer awareness may prevent side-channel mitigation transfer from one library to the other. This also includes libraries choosing to only partially mitigate the vulnerability, despite available secure alternatives.

*5.1.1 New contexts* Featuring a decade-old code-base, OpenSSL is particularly susceptible to this kind of vulnerability, as side-channel protection in one module might not be correctly ported to another. In particular, OpenSSL sets a CT flag on BIGNUMs marking secret data so that they can be manipulated using secure functions. Recent publications have shown that such *insecure-by-default* approaches are particularly error-prone [10, 33, 61, 120, 131].

**ECDSA.** Despite extensive research on mitigating scalar multiplication in ECDSA, Garcia et al. [60] identified exploitable usage of modular inversion with the vulnerable BEEA [5]. The vulnerable code path was taken because of a missing CT flag on the nonce.

**SM2.** The integration of the SM2 standard into OpenSSL did not inherit from lessons learned in ECDSA [120]. In particular, SM2 signature generation directly called the vulnerable wNAF scalar multiplication with no padding done on the scalar, allowing timing attacks [24]. Failure to set the appropriate CT flag also resulted in the modular inversion using BEEA.

**Key generation.** The RSA key generation process did not use side-channel protections, as single-trace attacks were thought to be impractical. Weiser et al. [131] proved this assumption false by exploiting secret-dependent control-flow in the BEEA used in key generation. Key generation was independently investigated through the test methodology Triggerflow [10, 68], finding additional issues stemming from the CT flag.

**Key parsing.** Similar vulnerabilities were found in OpenSSL and MbedTLS' key format handling [61], as key format standards leave a lot of flexibility to implementations. Differences in key format were found to induce different execution paths for a same operation, some calling vulnerable functions. This was the case for RSA key parsing/validation, and signatures for some elliptic curves. A similar problem was discovered earlier [9].

**SRP.** The missing CT flag steers OpenSSL implementation of SRP, a password-authenticated key exchange protocol, to an insecure variant of modular exponentiation using *square-and-multiply* [33].

**PRG.** Among the standard designs for pseudo-random generators (PRGs), `CTR_DRBG` generates a pseudo-random bit sequence using the AES cipher. Cohney et al. [49] investigated implementations of `CTR_DRBG` in multiple libraries, finding that the T-Table variant of AES was used despite its well-known vulnerabilities.

*5.1.2 New libraries* As the most popular cryptographic library [94], OpenSSL has received considerable attention from side-channel researchers. However, mitigations implemented in OpenSSL are not necessarily propagated to other libraries. MbedTLS RSA implementation uses the *sliding-window* method despite its vulnerability to cache attacks [88, 96]. Despite the implementation's attempt to balance branches by calling the same function in both branches of the square-and-multiply, Schwarz et al. [111] successfully exploited secret-dependent data access using a cache attack within an SGX enclave. A similar attack was also performed on the so-called left-to-right variant from Libgcrypt, featuring exponent blinding [121]. Still in SGX, the secret-dependent branch itself remains vulnerable to branch shadowing [86], or to attacks based on interactions between interrupts and instruction execution time [101]. Hassan et al. [122] illustrate this issue in the NSS library. While NSS' ECC code is forked from OpenSSL, mitigations such as nonce padding [37] are not implemented in NSS.

Some libraries implement *pseudo constant-time* instead of full constant-time to keep their code base easier to maintain. Such mitigations seek to address the Lucky 13 [11] attack by adding dummy MAC verification or random delays. Ronen et al. [109] demonstrate that they do not protect against cache attacks. Similarly, Ronen et al. [108] showed that padding oracles leading to Bleichenbacher-like attacks were still present in libraries such as MbedTLS, s2n, or NSS, due to early termination in padding verification.

### 5.2 Vulnerabilities in new functionalities

**Arithmetic functions.** Genkin et al. [64] targeted the X25519 key exchange implemented in Libgcrypt, finding that, while scalar multiplication is done using a *branchless* Montgomery ladder, the underlying finite field arithmetic functions were not constant-time. An early exit in the modular reduction function allowed the authors to mount a cache attack recovering the key. Aranha et al. [18] discovered a similar issue in OpenSSL branchless Montgomery Ladder used for ECDSA. When initializing the algorithm, a conditional swap is done depending on a secret value. While the swap itself is constant-time, the following finite field multiplication is not.

**Hash-to-element.** The Dragonfly handshake used in Wi-Fi authentication converts a shared password to an elliptic curve point, by computing the point coordinate derived from the password and a counter. When the obtained point is invalid, the counter is incremented and the operation repeated, thus creating a timing channel. While the standard suggests fixing the repetitions number by adding dummy ones, Vanhoef and Ronen [125] found that such mitigations were often incorrectly implemented, if at all, leaving implementations vulnerable to cache attacks [32].

**Post-Quantum Cryptography (PQC).** BLISS-B signature generation involves sampling a secret polynomial from a Gaussian distribution. Sampling methods based on pre-computed tables were shown to be vulnerable to cache attacks [97] and branch tracing attacks [119]. A common approach in PQC is to construct a cryptographic scheme secure against chosen plaintext attacks then making it secure against chosen ciphertext attacks using a generic transformation. Such transformation can lead to key-recovery attacks if not constant time [71].

## 5.3 Insights

The majority of recent publications reproduce vulnerabilities which have been long known, but in new context and libraries. Analysis should not focus on detecting constant-time violations in their code, but rather detect their incorrect use in the wider code-base. Test methodologies like TriggerFlow [68] are promising in this regard, and complementing them with fuzzing approaches could allow for wider exploration of cryptographic libraries. New vulnerabilities are not found in usual cryptographic primitives directly, but in newer protocols/schemes, or in lower-level utility functions. Detection tools thus need to be able to fully analyze programs, including utility functions, and scale to full protocol runs.

# 6 Tools considered

The rest of this paper is dedicated to comparing these approaches and characterizing them with regards to known vulnerabilities. We restrict ourselves to five tools: Abacus [20], Binsec/Rel [51], MicroWalk-CI [134], dudect [106], and ct-grind [85]. Together, they are representative of the diversity of methods of the literature and used in practice, with a mix of static and dynamic tools. The first three tools are from academic publications, while the last two are known to be used by developers [73]. Our selection is based on availability, ability to analyze binary code, scalability (from the claims in their publications), and guarantees provided by the tool.

## 6.1 Abacus

Abacus [20] first obtains an execution trace by running the target program with concrete inputs using the binary instrumentation framework Intel Pin [90]. Then, the SE engine *replays* the trace with the secret data set to symbolic values. Formulas are generated only for instructions manipulating secret values, otherwise concrete values are used. A control flow or memory access is represented as a function $f$ of the secret symbolic inputs $k$ and public concrete inputs $m$. The problem of whether an instruction violates CT is reformulated as whether $\exists k_1, k_2 \in K, f(k_1, m) \neq f(k_2, m)$, with $K$ the set of possible secrets. Contrary to prior work using DSE [127], Abacus does not use SMT solvers but randomly picks values for $k_1$ and $k_2$, as it is often sufficient to trigger the leakage.

In practice, the developer must indicate which variable is secret either through an annotation or writing a custom Pin tool. We noted a flaw in that Abacus only registers the first annotation and ignores the subsequent ones. As such, for our experiments, we wrote a Pin tool letting us mark as secret an arbitrary number of buffers of arbitrary length. Finally, Abacus computes an estimation of the number of bits leaked.

## 6.2 Binsec/Rel

Binsec/Rel [51] uses SE for bug-finding and bounded-verification of constant-time at binary level. It lifts the target binary to an IR, performs SE along all program paths, and checks that control-flow and memory accesses do not depend on secret. More precisely, Binsec/Rel uses *relational* SE to model two executions of a program in the same SE instance, with the same public input but distinct secret inputs. SMT queries allow checking whether these pairs of path can diverge on control-flow or memory addresses. If a query is satisfiable, the program is insecure and the solver returns a counter-example triggering the vulnerability. Conversely, if all queries are unsatisfiable and the exploration is exhaustive, the program is secure. In practice, these queries are expensive, so Binsec/Rel uses optimizations to spare unsatisfiable queries, making it scale on secure programs. However, satisfiable queries may remain a bottleneck on insecure programs.

**Binsec/Rel2.** The Binsec/Rel team has recently developed a second version, Binsec/Rel2 [27], with better general performance and architecture support (based on the latest version of the symbolic engine), easier set up (core-dump based initialization), and dedicated optimization for satisfiable queries—based on evaluation over pre-chosen concrete values.

## 6.3 MicroWalk-CI

MicroWalk [133] records multiple executions of the target function under different inputs using Intel Pin. The recorded trace contains branch targets and memory addresses encountered during the execution. Mutual information scores are then computed between the leakage trace and the input set, giving a quantification of the number of input bits leaked. MicroWalk offers various MI score granularity/performance tradeoffs, ranging from whole program MI—giving a coarse leakage quantification—to single instructions—for exact leakage localization. MicroWalk-CI [134] adds support for Javascript. Special attention has been paid to the framework's usability, with human readable reports compatible with CI features.

## 6.4 dudect

dudect [106] detects timing leakage through repeated timing measurements and comparison using a statistical test. The binary is executed under two different classes of secret inputs: one set to constant values and one set to values randomly selected before each measurement. The execution timings of the target function are then recorded by probing CPU cycle counters. Measurements above a certain percentile $p$ are assumed to be noise and are removed. An online t-test is then applied to determine whether the two distributions are distinguishable, with leakage being reported

if the value passes a predefined threshold. While this approach is simple and easy to deploy, it only gives guarantees *up to a certain number of measurements*.

### 6.5 ctgrind

ctgrind [85] uses Valgrind's memory error detector, Memcheck, to detect potential side-channel leakages. As Memcheck already detects branches and memory accesses computed on uninitialized memory, side-channel vulnerabilities can be found by marking secret variables as undefined, through a specific code annotation. Internally, Memcheck detects errors by shadowing every bit of data manipulated by the program with a *definedness* bit V [113], propagated throughout the execution similarly to a taint analysis and checked when computing an address or a jump.

Applied to side-channel analysis however, this approach yields a considerable number of false positives, as errors unrelated to secret values are also reported. Still, ctgrind is particularly popular in cryptographic libraries for its simplicity and ease of deployment: ctgrind is used by 5 out of the 6 cryptographic libraries performing CT tests as reported by [73].

## 7 Unified benchmark

To fairly compare the scalability and vulnerabilities reported by existing approaches (**RQ1**), we create a unified benchmark, comprised of representative cryptographic operations from 3 libraries, totaling 25 benchmarks. Table 2 presents our results.

### 7.1 Description and setup

**Libraries.** We include OpenSSL [4], as it is one of the most popular cryptographic library [94] and has a long history of side-channel vulnerabilities. MbedTLS [58] is another popular library, particularly for embedded targets. Its maintainers consider side-channel attacks in their threat model, though it is a work in progress relevant mainly for new code. BearSSL [99] is a smaller library with an emphasis on ease of deployment and security. In particular, the author makes clear claims on which functions are constant-time, which helps us establish a *ground truth* for our benchmark. For each of these libraries we pick the latest versions at the times of experiment (3.0.5, 3.1 and 0.6 respectively), and compile the libraries both as static 32 and 64-bits objects, to meet the detection tool requirements. While we keep library configurations close to their defaults, for MbedTLS, we disable support for VIA PadLock instructions and AES table generation code.

**Algorithms.** We chose widely-used cryptographic operations from both symmetric and asymmetric schemes. We target AES encryption, in both CBC and the authenticated GCM mode, as well as the Poly1305-Chacha20 AEAD scheme. We target RSA decryption with both PKCS#1 v1.5 and OAEP padding scheme. ECDSA signature generation using P-256 and EdDSA signature generation using Curve25519 are also included.

**Implementations.** A library can include different implementations of a single algorithm with differing impacts on side-channel leakage, notably for AES. For OpenSSL we include the vulnerable T-table implementation and the secure vector permutation (VP) implementation. For BearSSL, we include its T-table implementation and its constant-time implementation, based on a bit-slicing (BS) approach. Such implementations also exist for OpenSSL, but not for our 32-bits configuration. Finally, we add a benchmark using OpenSSL's EVP API, which is the intended entry point for developers and dynamically selects the best implementation available.

**Limits.** These common operations are not all supported in the same way. BearSSL only supports RSA decryption with OAEP padding, where only OpenSSL supports EdDSA. While we favor deterministic ECDSA as it derives the secret from the key without any PRNG, OpenSSL does not support it at the time of our experiments [3]. We thus use the non-deterministic version.

**Benchmark design.** We limit ourselves to simple test harnesses, limiting the number of auxiliary operations before the target cryptographic operation in order to avoid introducing extra issues in terms of scalability, instruction support, or non-determinism. Only the input of the target operation are marked as secret. For symmetric ciphers we mark the shared key as secret, for RSA the secret key parameters (including the CRT parameters), for elliptic curves the secret scalar. For each benchmark, the arrays containing secret values are also marked as such at the beginning of the main. While ideally we would run all the selected detection tools on the same binary for a given benchmark, the tools have conflicting requirements. As such, for each benchmark we produce both a 32-bits and a 64-bits version, the latter being used only by Microwalk.

**Evaluation protocol.** We compare the numbers of vulnerabilities found by each tool and the running times of the analysis. All experiments are run on a laptop equipped with an i7-8650U processor running Ubuntu 20.04 LTS. For Abacus, Microwalk and both versions of Binsec/Rel, the vulnerability count is taken directly from the tool output, corresponding to the number of unique leaky instructions. For ctgrind, some post-processing is needed on the output.

First, ctgrind reports a single leaking instruction multiple times if it is reached in different contexts, so we deduplicate the reported error count. Second, as ctgrind is based on Valgrind, errors unrelated to constant-time are also be reported. For example, on statically linked programs, Valgrind reports problems from glibc functions [2]. To remedy this, we generate a suppression file by first running the program without annotating the secret. As a result, errors independent of the secret are not reported. The running time of the analysis is measured as the total time from first invoking the tool to its exit. For Binsec/Rel, this includes generation of the coredump when needed. For Abacus, this includes the time needed to record a trace using Pin, as such our running times are generally longer than those in the original publication [20]. For dudect, computations of the t-test score are done in batches of 5000 measurements, repeated until the analysis either finishes or times out. New inputs are generated for each measurement. We additionally thrash the cache by accessing a large array between each run of the targeted algorithm to limit cache side-effects. Microwalk is set up to generate 16 traces, as recommended by the authors [134]. We supply for each trace a different set of inputs and use the number of unique leaking instructions reported by the tool.

**Table 2: Vulnerability detection. #V: number of reported vulnerabilities, S: status (● secure; ○ insecure; ◐ unknown), t: time to first bug in seconds, T: analysis time in seconds.**

| | Binsec/Rel | | | | Binsec/Rel2 | | | | Abacus | | | ctgrind | | | Microwalk | | | dudect | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmarks | #V | S | t (s) | T (s) | #V | S | t (s) | T (s) | #V | S | T (s) | #V | S | T (s) | #V | S | T(s) | S | T (s) |
| AES-CBC-bearssl (T) | 20 | ○ | 0.05 | ⧗ | 36 | ○ | 0.02 | 0.10 | 36 | ○ | 3.65 | 36 | ○ | 0.16 | 36 | ○ | 1.39 | ○ | 100.51 |
| AES-CBC-bearssl (BS) | 0 | ● | – | 16.77 | 0 | ● | – | 0.31 | 0 | ◐ | 10.69 | 0 | ◐ | 0.17 | 0 | ◐ | 1.55 | ◐ | ⧗ |
| AES-GCM-bearssl (T) | 20 | ○ | 0.05 | ⧗ | 36 | ○ | 0.02 | 0.22 | 36 | ○ | 6.23 | 36 | ○ | 0.18 | 36 | ○ | 1.51 | ○ | 8.69 |
| AES-GCM-bearssl (BS) | 0 | ● | – | 53.32 | 0 | ● | – | 0.48 | 0 | ◐ | 14.79 | 0 | ◐ | 0.17 | 0 | ◐ | 1.73 | ◐ | ⧗ |
| AES-CBC-mbedtls (T) | 20 | ○ | 0.1 | ⧗ | 68 | ○ | 0.03 | 0.17 | 68 | ○ | 5.73 | 68 | ○ | 0.19 | 68 | ○ | 1.54 | ○ | 0.63 |
| AES-GCM-mbedtls (T) | 20 | ○ | 0.27 | ⧗ | 84 | ○ | 0.05 | 0.4 | 84 | ○ | 261.76 | 76 | ○ | 0.19 | 71 | ○ | 1.90 | ○ | 0.89 |
| AES-CBC-openssl (EVP) | 0 | ◐ | – | ⧗ | 0†‡ | ◐ | – | 21.92 | 0 | ◐ | 103.59 | 0 | ◐ | 0.66 | 9 | ○ | 5.35 | ◐ | ⧗ |
| AES-GCM-openssl (EVP) | 0 | ◐ | – | ⧗ | 0†‡ | ◐ | – | 21.19 | 0 | ◐ | 104.27 | 70 | ○ | 0.71 | 8 | ○ | 5.66 | ◐ | ⧗ |
| AES-CBC-openssl (T) | 20 | ○ | 0.05 | ⧗ | 36 | ○ | 0.02 | 0.16 | 36 | ○ | 3.02 | 36 | ○ | 0.18 | 20 | ○ | 1.38 | ◐ | ⧗ |
| AES-CBC-openssl (VP) | 0† | ◐ | – | 0.06 | 0‡ | ● | – | 0.59 | 0 | ◐ | 1.01 | 0 | ◐ | 0.19 | 0 | ◐ | 1.68 | ◐ | ⧗ |
| PolyChacha-bearssl (CT) | 0 | ● | – | 9.72 | 0 | ● | – | 0.18 | 0 | ◐ | 1.3 | 0 | ◐ | 0.16 | 0 | ◐ | 1.43 | ◐ | ⧗ |
| PolyChacha-bearssl (SSE2) | 0† | ◐ | – | 0.05 | 0 | ● | – | 0.11 | 0 | ◐ | 1.17 | 0 | ◐ | 0.15 | 0 | ◐ | 1.34 | ◐ | ⧗ |
| PolyChacha-mbedtls | 0 | ● | – | 18.62 | 0 | ● | – | 0.49 | 0 | ◐ | 5.02 | 0 | ◐ | 0.18 | 0 | ◐ | 2.95 | ◐ | ⧗ |
| PolyChacha-openssl (EVP) | 0 | ◐ | – | ⧗ | 0‡ | ● | – | 21.55 | 💣* | ◐ | 100.19 | 0 | ◐ | 0.67 | 8 | ○ | 5.41 | ◐ | ⧗ |
| Chacha20-openssl | 0 | ● | – | 0.77 | 0 | ● | – | 0.09 | 0 | ◐ | 3.77 | 0 | ◐ | 0.15 | 0 | ◐ | 1.36 | ◐ | ⧗ |
| Poly1305-openssl | 0 | ● | – | 0.26 | 0‡ | ● | – | 0.35 | 💣* | ◐ | 1.19 | 0 | ◐ | 0.17 | 0 | ◐ | 1.70 | ◐ | ⧗ |
| RSA-bearssl (OAEP) | 0 | ◐ | – | ⧗ | 2 | ○ | 0.03 | ⧗ | 💣* | ◐ | 356.41 | 87 | ○ | 0.57 | 0 | ◐ | 146.52 | ◐ | ⧗ |
| RSA-mbedtls (PKCS) | 1 | ○ | 1.60 | ⧗ | 5‡ | ○ | 0.21 | ⧗ | 💣* | ◐ | 2193.2 | 39 | ○ | 0.96 | 137 | ○ | 826.23 | ◐ | ⧗ |
| RSA-mbedtls (OAEP) | 1 | ○ | 1.46 | ⧗ | 5‡ | ○ | 0.21 | ⧗ | 💣* | ◐ | 2203.89 | 48 | ○ | 0.98 | 137 | ○ | 847.27 | ◐ | ⧗ |
| RSA-openssl (PKCS) | 1 | ○ | 26.50 | ⧗ | 1‡ | ○ | 0.44 | ⧗ | 0 | ◐ | 551.72 | 321 | ○ | 1.32 | 46 | ○ | 52.06 | ○ | 618.73 |
| RSA-openssl (OAEP) | 1 | ○ | 29.02 | ⧗ | 1‡ | ○ | 0.46 | ⧗ | 💣* | ◐ | 535.91 | 546 | ○ | 1.73 | 61 | ○ | 59.90 | ○ | 771.3 |
| ECDSA-bearssl | 2 | ○ | 1.82 | ⧗ | 2 | ○ | 0.84 | ⧗ | 0 | ◐ | 246.62 | 7 | ○ | 0.41 | 0 | ◐ | 109.31 | ◐ | ⧗ |
| ECDSA-mbedtls | 0 | ◐ | – | 0.54 | 0†‡ | ◐ | – | 0.19 | 0 | ◐ | 467.11 | 50 | ○ | 0.81 | 132 | ○ | 314.26 | ○ | 224.30 |
| ECDSA-openssl | 0 | ● | – | 2252.36 | 1‡ | ○ | 51. | ⧗ | 0 | ◐ | 202.72 | 14 | ○ | 1.22 | 28 | ○ | 13.76 | ◐ | ⧗ |
| EdDSA-openssl | 0 | ◐ | – | ⧗ | 0‡ | ● | – | 25.63 | 0 | ◐ | 59.87 | 0 | ◐ | 1.03 | 8 | ○ | 9.86 | ◐ | ⧗ |

💣*: the tool crashed. ⧗: the analysis timed-out after 3600 s. †: the analysis terminated early (e.g., unsupported instructions). ‡: the analysis starts from a core-dump (Binsec/Rel2 only)

## 7.2 Results and discussion

Table 2 reports, for each tool, the running time of the analysis (T) and the number of unique vulnerabilities (#V) found when appropriate. We also report in column S whether the analysis finds the overall program is secure (●), insecure (○) or if no statement can be made (◐). As dynamic tools only model a limited number of executions, they can only report ○ and ◐. Static tools can report ● and ○, reporting ◐ when the analysis times out or stops early without finding vulnerabilities. For Binsec/Rel we also report the time required to find a first vulnerability (t).

**Symmetric primitives.** For simpler symmetric cryptography such as AES-CBC and authenticated schemes like AES-GCM or Poly1305-Chacha20, tools that report a number of vulnerabilities tend to agree with each other. In general, the tools scale reasonably well on such primitives. Implementations using constant-time techniques such as vector permutations (VP) or bit-slicing (BS) are reported as secure while those based on table lookups (T) yield a consistent number of vulnerabilities. dudect can determine that these implementations are insecure, but the OpenSSL T-table implementation as it preloads tables in the cache before accessing them.

While these numbers are consistent, there are two notable exceptions. **First**, ctgrind and Microwalk show less vulnerabilities in MbedTLS AES-GCM implementation than other tools. For the former, this discrepancy stems from a bug within Valgrind, where upon encountering two consecutive vulnerable memory accesses, the error reporting for one will "shadow" the other. This is relevant in this benchmark as MbedTLS' GHASH function is implemented using 64-bit integers which, in 32-bit, are compiled into two `mov` instructions. We have not encountered this issue in other benchmarks. For the latter, this could be explained by the fact that Microwalk does not report CT violation where the leakage quantified is too small, whereas tools not providing such quantification will report these in any case. **Second**, in the OpenSSL's EVP implementation of AES-GCM, ctgrind reports a large number of vulnerabilities while other tools do not. For Binsec/Rel2, this is unsurprising as its analysis stops upon reaching unsupported AES-NI instructions used by the EVP implementation. The vulnerabilities reported by ctgrind are all located in the GHASH implementation, which employs table look-ups to perform multiplications in $GF(2^{128})$, a potential vulnerability identified in [77]. Microwalk reports a few vulnerabilities as well, located in functions used to select which implementations to run

**Asymmetric primitives.** In the case of more complex asymmetric cryptography, results vary highly between tools. For Abacus, no vulnerabilities are found when the analysis finishes, however, we observe multiple crashes. Analyses often time out on both Binsec/Rel and Binsec/Rel2, only finding a handful of vulnerabilities. For OpenSSL and MbedTLS these vulnerabilities are located

in BIGNUM conversion functions called before the target cryptographic operation. Such functions have been found to be vulnerable before [130, 141], we see here that they are also a source of scalability issues. For dudect, the need to generate inputs for each measurement is a significant bottleneck, in particular for RSA where key generation is a costly operation. As a result of slower key generation in MbedTLS and BearSSL, the respective RSA benchmarks time out before finishing a single set of measurements. In the case of ctgrind, analyses finishes very quickly, however the large number of vulnerabilities found makes interpreting these results complicated, with most vulnerabilities being found in BIGNUM functions. Microwalk provides similarly high vulnerability counts, though with analysis times in the order of minutes. We note that among these vulnerabilities we find again some related to implementation selection, in particular for OpenSSL.

**Unsupported instructions.** Unsupported instructions can have a crucial impact on the analysi' results. The vulnerable GHASH implementation is rarely reached in practice, as OpenSSL steers the execution to constant-time variants making uses of carry-less multiplication (`CLMUL`) instructions. However Valgrind emulates the results of `cpuid` to only include instruction sets it supports. Its analysis thus does not reach the `CLMUL` implementation, instead defaulting to a vulnerable function. The results we obtained with Abacus contrast with the authors' own RSA and ECDSA benchmarks [20]. While the authors did not disclose which configurations were used to compile the libraries, disassembling their benchmarks shows that, at least for OpenSSL, assembly implementations were disabled. Such implementations are included in our benchmark, yielding SIMD instructions unsupported by Abacus. While in some cases these cause Abacus to crash, it is also possible they lead to under-tainting, explaining the low vulnerability counts.

**False positives.** Both ctgrind and Binsec/Rel2 report violations in RSA-bearssl and ECDSA-bearssl, which is surprising given BearSSL constant-time policies. A closer inspection reveals that the RSA implementation computes the actual bit length of RSA primes by counting the number of nonzero bytes, triggering secret-dependent control-flow, while ECDSA performs an early exit if the key is not well-defined. While these violations are technically true CT violations, they are not exploitable vulnerabilities. Microwalk on the other hand does not report these CT violations as they do not result in differences in traces for the inputs we used. In both cases, Binsec/Rel2 reports a subset of the vulnerabilities reported by ctgrind, which can be explained by a different notion of unique vulnerability between both tools. Indeed, Binsec/Rel2 only reports a single violation when leaking the same data on different locations, whereas ctgrind reports multiple locations leaking the same data. Finally, violations are reported in RSA and ECDSA implementations of OpenSSL and MbedTLS, however these implementation use blinding, making the violations unexploitable in practice. This non-determinism could introduce differences in traces which can be reported as vulnerabilities by tools like Microwalk. We note however that Microwalk features a way to *spoof* the outputs of `rdrand` which can alleviate this problem.

**Leakage quantification.** We did not report the numbers given by Abacus and MicroWalk, as there is no standard way of computing leakage quantification or determining their severity. Moreover, leakage quantification is far from a practical notion of exploitability, as even a single bit of leakage or less can be exploited [18].

**Improvements over Binsec/Rel.** Binsec/Rel may struggles to fully analyze insecure programs, even simpler ones like AES. This issue is alleviated in Binsec/Rel2 thanks to the chosen value optimization. Binsec/Rel2 is able to finish analyzing symmetric primitives in times comparable to even dynamic tainting approaches like ctgrind, finding the same number of vulnerabilities. While scalability remains an issue for asymmetric primitives, Binsec/Rel2 is able to analyze the OpenSSL implementation of Ed25519 faster than a dynamic method like Abacus. The ability to start the analysis from a coredump solves initialization issues (e.g., global function pointers) that cannot be avoided in Binsec/Rel, while limiting the amount of instrumentation required.

## 8 Case-study: vulnerability validation

We now employ the tools described in Section 6 to determine if the vulnerabilities in Section 5 could have been discovered automatically (**RQ2**), and determine features that are missing from existing frameworks to find them (**RQ3**).

### 8.1 Description and setup

**Targeted vulnerabilities.** We consider vulnerabilities from three publications: [10] focused on RSA key generation in OpenSSL 1.0.2k, [61] focused on key format handling in OpenSSL 1.1.1a, and MbedTLS 2.18.1 (in particular P256 keys for the former and RSA keys for the latter), and [64] focused on ECDH decryption in Libgcrypt 1.7.6. Following our typology (Section Section 5), the first two papers describe vulnerabilities stemming from functions known to be vulnerable, but accidentally used in a new context, while the third one represents a new vulnerability.

**Benchmark design.** While we are interested in checking for CT violations in these vulnerable functions, a library developer without prior knowledge is unlikely to analyze the right function (e.g., GCD, modular exponentiation). More realistically, such developer would instead check the larger context in which this function is used (e.g., RSA key generation). As such, we are interested in detecting CT violations not just in the vulnerable function itself, but also in its calling context, where detection tools might struggle with scalability, or other usability issues. We thus run the detection tools on simple programs calling the vulnerable functions directly and the higher-level operation.

**Evaluation protocol.** We compare the number of vulnerabilities and running times of the analysis from Binsec/Rel2, Abacus, ctgrind and dudect with their configurations from Section 7.

### 8.2 Results and discussion

Table 3 reports, for each tool, the running time of the analysis (T) and the number of vulnerabilities (#V) found when available. Abacus report vulnerabilities only upon finishing the analysis, and thus reports none when there is a timeout. Additionally for each tool, we report whether a vulnerability useful for the publications' attacks is reported (V). The higher-level operations are listed in bold, while the vulnerable functions used are listed immediately below the corresponding operation.

**Table 3: Vulnerability detection. V: whether the right vulnerability is found #V: number of reported vulnerabilities, S: status (● secure; ○ insecure; ◐ unknown), T: analysis time in seconds.**

| | Binsec/Rel2 | | | | Abacus | | | | ctgrind | | | | Microwalk | | | | dudect | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmarks | V | #V | S | T (s) | V | #V | S | T (s) | V | #V | S | T (s) | V | #V | S | T (s) | S | T (s) |
| **P256 sign (OpenSSL)** | | 0 | ● | 16.09 | | 💣* | ◐ | 5.69 | | 0 | ◐ | 0.75 | ✓ | 33 | ○ | 71.44 | ◐ | – |
| wNAF multiplication | | 5 | ○ | ⧗ | | – | ◐ | ⧗ | ✓ | 222 | ○ | 0.54 | ✓ | 35 | ○ | 62.55 | ○ | 24.5 |
| **RSA valid. (MbedTLS)** | | 5 | ○ | ⧗ | | 0 | ◐ | 490.01 | ✓ | 31 | ○ | 0.40 | ✓ | 41 | ○ | 278.94 | ○ | 2205.07 |
| GCD | | 5 | ○ | ⧗ | | 0 | ◐ | 37.74 | | 14 | ○ | 0.21 | ✓ | 8 | ○ | 22.96 | ○ | 4.26 |
| modular inversion | | 3 | ○ | ⧗ | | 0 | ◐ | 242.1 | ✓ | 19 | ○ | 0.24 | ✓ | 17 | ○ | 141.82 | ○ | 17.7 |
| **RSA keygen (OpenSSL)** | | 0 | ● | 0.17 | | 💣* | ◐ | 8.66 | | 0 | ◐ | 6.36 | ✓ | 123 | ○ | 842.02 | ◐ | – |
| GCD | ✓ | 3 | ○ | ⧗ | | – | ◐ | ⧗ | | 53 | ○ | 0.19 | ✓ | 4 | ○ | 3.61 | ○ | 0.7 |
| modular exponentiation | | 1 | ○ | ⧗ | ✓ | 4 | ○ | 110.73 | ✓ | 6 | ○ | 0.24 | ✓ | 25 | ○ | 13.52 | ○ | 6.96 |
| modular inversion | | 1 | ○ | ⧗ | | – | ◐ | ⧗ | ✓ | 115 | ○ | 0.21 | ✓ | 15 | ○ | 5.96 | ○ | 1.69 |
| **ECDH decryption (Libgcrypt)** | | 2 | ○ | ⧗ | | 0 | ◐ | 386.76 | ✓ | 359 | ○ | 1. | | 34 | ○ | 146.50 | ○ | 89.46 |
| modular reduction | | 0 | ◐ | ⧗ | | 0 | ◐ | 2.35 | ✓ | 9 | ○ | 0.22 | | 0 | ◐ | 1.59 | ○ | 0.41 |

**General results.** In general, Binsec/Rel2, ctgrind, and dudect report the same programs as secure (*i.e.*, P256 sign and RSA keygen), and the rest as insecure, with the exception of the modular reduction in Libgcrypt, for which Binsec/Rel2 times-out without finding vulnerabilities. Microwalk however finds almost all programs insecure, with error counts wildly different from ctgrind. In contrast, Abacus only reports a single program as insecure. This is possibly due to unsupported instructions, once again limiting the tool ability to properly taint secret data. Additionally, the analysis will times out in multiple cases, returning no results. The ability to output violations as soon as they are found would, thus, greatly improve usability. In general, we find that ctgrind and Microwalk in particular are able to find most of the relevant vulnerabilities in both the target functions and their calling contexts. While the relevant vulnerabilities can generally be found, the high number of reported vulnerabilities, in particular for ctgrind, complexifies the interpretation of these results. We note that many of these vulnerabilities stem from BIGNUM manipulation functions. These CT violations are often already known by developers, and so, as pointed out by Jancar et al. [73], the ability to ignore violations in parts of the code would represent a clear usability improvement. Additionally, both ctgrind and Microwalk report vulnerabilities found in standard functions (e.g., malloc). These are usually ignored by both tools, however they are included as our binaries are statically linked. Contrasting with ctgrind, Binsec/Rel2 found few vulnerabilities. We note that the analysis often gets stuck early in the program, during BIGNUM conversion steps, and times out. Specifically, benchmarks involving GCD computations and modular inversion are susceptible to path explosion, a common limitation of static SE, which leads to unexplored program behaviors.

**Implicit flows.** Implicit flows happen when the value of a variable may differ depending on secret-dependent control flow. While none of the tools we considered explicitly tracks implicit information flows, these can produce trace difference which Microwalk can detect. Our benchmarks shows that ignoring them leads to an underestimation of the number vulnerabilities in cryptographic programs. For example, MbedTLS implements a comparison function in which the return value depends on whether a secret-dependent branch is taken (a typical case of an implicit flow). MbedTLS GCD function uses the result of this comparison function in a (secret-dependent) conditional jump. As a result, ctgrind only reports the secret-dependent branch in the comparison function as vulnerable, but not the one in the GCD computation. While a CT violation is still reported, a developer is not able to fully appreciate how vulnerable the program is without taking into account these implicit flows. Microwalk on the other hand, is able to detect the difference and correctly reports the conditional jump in GCD as vulnerable. This illustrates a clear advantage that trace comparison methods have over other dynamic methods.

**Internal secrets.** The way the secret is utilized in some operations can pose problems for detection tools. For RSA key generation, there is by definition, no initial values to mark as secret. For P256 signature generation, the secret exploited in publications is the cryptographic nonce generated internally. Other operations making use of PRNG are also affected similarly. In both of these cases our "black-box" experimental setup does not allow us to mark the secret appropriately and as such, the detection tools we used cannot find vulnerabilities. In practice, it is possible to mark such secrets by deviating from our setup. For example, the authors of Abacus wrote a special pintool to mark the cryptographic nonce as secret, Surprisingly, Microwalk is able to find the vulnerabilities in these functions despite no clear way to correlate the traces with the inputs. We conjecture that this is due to trace differences induced by the randomness of the generation process. but their solution lacks generality. Another option is to add annotations within a library's source code, something that developers are generally opposed to. For processes involving random number generation, a tool could consider the output of the entropy source as an input to the target function.

## 9 Recommendations

We now formulate recommendations to the research community and to cryptographic library developers based on the insights obtained with our experimental evaluation.

## 9.1 Recommendations to research community

**R1. Support of SIMD instructions.** Research prototypes like Abacus often only support basic x86 instructions. However, as shown in Section 7, libraries heavily use SIMD instruction sets (e.g., AVX2), now commonly supported in CPUs. These implementations are now typically selected in priority. As such, supporting SIMD instructions in detection tools is now crucial, and evaluating a tool on implementations specifically selected to *not* use SIMD instructions may mislead developers on the tool usability.

**R2. Support of implicit information flows.** Future tools should investigate the possibility of detecting CT violations stemming from implicit information flows, in particular for static tools and dynamic tools using a single trace. We argue that by ignoring them, a detection tool can underestimate the amount of vulnerabilities, giving developers a false sense of security. Yet, considering implicit flows is challenging and can impact scalability. Only three tools explicitly support implicit flows [74, 107, 141], while trace-comparison-based dynamic tools might only provide a partial support.

**R3. Support of internal secrets.** The community should investigate side-channel vulnerabilities in operations where the secret is generated internally, such as key generation and PRNG [10, 49, 101]. None of the tools in our classification explicitly support them, although in practice we found Microwalk promising in this regards. The ability to set the output of an entropy source, beyond a clear usability advantage, could be extended to be marked as secret. Recently, a test methodology to detect usage of vulnerable functions in key generation was proposed [68], however it requires knowing vulnerable points in advance.

**R4. Support for randomization-based defense.** Blinding introduces randomization during computations to hinder inference of secrets via side-channels [75]. It poses an additional challenge for detection tools as the leakage does depend on the secret, but not in an exploitable way, leading to false positives. Only 3 out of the 34 tools we considered in Table 1 support blinding.

**R5. Usage of a standardized benchmark.** We recommend the community to adopt a standardized benchmark of cryptographic implementations to evaluate side-channel detection tools. Such benchmark would facilitate comparing detection tools to one another, in particular in terms of usability and scalability. We thereby propose to the community our benchmark, which can be readily adopted and extended in the future.

**R6. Improve usability.** As pointed out in Section 8, reporting results before the analysis end is crucial for benchmarks that times out. We also echo a point made in [73]: being able to ignore vulnerabilities from some parts of the code helps interpreting the results, particularly for approaches like ctgrind. For static tools, the ability to write stubs adding limited support for system calls and dynamic allocation was essential in our experiments with Binsec/Rel, as we cannot expect developers to avoid using them. The ability to start the analysis from a coredump (as in Binsec/Rel2) also greatly simplifies the instrumentation needed to initialize the analysis.

## 9.2 Recommendations to developers

**R7. Make libraries more analysis friendly.** Static analysis tools often require more instrumentation than dynamic ones. In particular, the selection of implementations at run-time poses a challenge. While for experiments using Binsec/Rel2 we were able to bypass this issue by initializing the memory from a coredump, we can only analyze one implementation and have little control over which one is chosen. We recommend that libraries expose in their API, at least for testing purposes, functions to directly call different implementations, as done for example by OpenSSL AES implementations.

## 10 Conclusion

We surveyed and classified state-of-the-art side-channel detection tools, and propose a unified benchmark allowing fair experimental comparison. Our findings show that existing tools can still struggle to analyze complex primitives and can miss vulnerabilities. We issued recommendations for the research and developer community to improve existing tools and encourage their usage.

Beyond these recommendations we note that the research communities for microarchitectural and physical side-channel detection both evolved independently, with little overlap [39]. With the advent of frequency scaling side-channels like Hertzbleed [129] extracting a timing side-channel from power consumption, this boundary is blurred. To properly address such vulnerabilities, the two communities should focus on bridging the gap between them, borrowing insights from physical side-channel detection and move beyond the constant-time model.

## Acknowledgments

This work benefited from the support of the ANR-19-CE39-0007 MIAOUS, ANR-20-CE25-0009 TAVA, PEPR PP Secureval and PEPR PP Rev projects.

## References

[1] [n. d.]. BearSSL - Constant-Time Crypto. https://bearssl.org/constanttime.html.
[2] [n. d.]. Bug 200535 - Valgrind flags lots of uninitialized locations with programs compiled with "-static". https://bugzilla.redhat.com/show_bug.cgi?id=200535.
[3] [n. d.]. Implement deterministic ECDSA sign (RFC6979). https://github.com/openssl/openssl/pull/18809.
[4] [n. d.]. OpenSSL. https://www.openssl.org/.
[5] O. Aciiçmez, S. Gueron, and J.-P. Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *IMACC*.
[6] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys Via Branch Prediction. In *CT-RSA*.
[7] Johan Agat. 2000. Transforming Out Timing Leaks. In *POPL*.
[8] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *EUROCRYPT*.
[9] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *S&P*.
[10] A. Cabrera Aldaya, C. Pereida García, L. M. Alvarez Tapia, and B. B. Brumley. 2019. Cache-Timing Attacks on RSA Key Generation. *TCHES* (2019).
[11] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *S&P*.
[12] J. Bacelar Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *CCS*.
[13] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX*.
[14] J. Bacelar Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P-Y. Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *S&P*.
[15] J. Bacelar Almeida, M. Barbosa, J. Sousa Pinto, and B. Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.* (2013).

[16] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *S&P*.

[17] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *PLDI*.

[18] D. F. Aranha, F. Rodrigues Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *CCS*.

[19] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2018. SideTrail: Verifying Time-Balancing of Cryptosystems. In *VSTTE*.

[20] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. 2021. Abacus: Precise Side-Channel Analysis. In *ICSE*.

[21] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. 2021. SoK: Computer-Aided Cryptography. In *S&P*.

[22] G. Barthe, G. Betarte, J. Diego Campo, C. Daniel Luna, and D. Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *CCS*.

[23] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Math. Struct. Comput. Sci.* 21, 6 (2011), 1207–1252.

[24] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom. 2014. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *CHES*.

[25] Daniel J Bernstein. 2005. Cache-Timing Attacks on AES.

[26] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *LATINCRYPT*.

[27] BinsecRel2. 2023. BinsecRel2. https://binsec.github.io/releases/binsec/2023/02/14/binsec-0.7.1.html.

[28] Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying Constant-Time Implementations by Abstract Interpretation. In *ESORICS*.

[29] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *CRYPTO*.

[30] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*.

[31] 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*.

[32] D. De Almeida Braga, P.-A. Fouque, and M. Sabt. 2020. Dragonblood is Still Leaking: Practical Cache-based Side-Channel in the Wild. In *ACSAC*.

[33] D. De Almeida Braga, P.-A. Fouque, and M. Sabt. 2021. PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild. In *CCS*.

[34] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. 2018. Symbolic path cost analysis for side-channel detection. In *ISSTA*.

[35] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *S&P*.

[36] B. B. Brumley and R. M. Hakala. 2009. Cache-Timing Template Attacks. In *ASIACRYPT*.

[37] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *ESORICS*.

[38] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Comput. Networks* 48, 5 (2005), 701–716.

[39] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. 2022. SoK: Design Tools for Side-Channel-Aware Implementations. In *AsiaCCS*.

[40] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX@SOSP*.

[41] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*.

[42] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[43] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. 2018. Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers. In *CCS*.

[44] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*.

[45] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*.

[46] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *PLDI*.

[47] S. Chattopadhyay and A. Roychoudhury. 2018. Symbolic Verification of Cache Side-Channel Freedom. *Trans. Comput. Aided Des. Integr. Circuits Syst.* (2018).

[48] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *CCS*.

[49] S. Cohney, A. Kwong, S. Paz, D. Genkin, N. Heninger, E. Ronen, and Y. Yarom. 2020. Pseudorandom Black Swans: Cache Attacks on CTR_DRBG. In *S&P*.

[50] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*.

[51] L.-A. Daniel, S. Bardin, and T. Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *S&P*.

[52] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2022. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Transactions on Privacy and Security* (2022).

[53] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (1977), 504–513.

[54] C. Disselkoen, S. Cauligi, D. Tullsen, and D. Stefan. 2020. Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork. In *TECHCON*.

[55] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security*.

[56] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *PLDI*.

[57] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*.

[58] Trusted Firmware. [n. d.]. Mbed TLS. https://www.trustedfirmware.org/projects/mbed-tls/.

[59] CRoCS: Centre for Research on Cryptography and Security. [n. d.]. Constant-timeness verification tools. https://crocs-muni.github.io/ct-tools/.

[60] Cesar Pereida García and Billy Bob Brumley. 2017. Constant-Time Callees with Variable-Time Callers. In *USENIX Security*.

[61] C. Pereida García, S. ul Hassan, N. Tuveri, I. Gridin, A. Cabrera Aldaya, and B. B. Brumley. 2020. Certified Side Channels. In *USENIX Security*.

[62] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27.

[63] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *CRYPTO*.

[64] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *CCS*.

[65] Jovan Dj. Golic and Christophe Tymen. 2002. Multiplicative Masking and Power Analysis of AES. In *CHES*.

[66] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*.

[67] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.

[68] I. Gridin, C. Pereida García, N. Tuveri, and B. B. Brumley. 2019. Triggerflow: Regression Testing by Advanced Execution Path Inspection. In *DIMVA*.

[69] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *ICISC*.

[70] D. Gruss, R. Spreitzer, and S. Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*.

[71] Qian Guo, Thomas Johansson, and Alexander Nilsson. 2020. A Key-Recovery Timing Attack on Post-quantum Primitives Using the Fujisaki-Okamoto Transformation and Its Application on FrodoKEM. In *CRYPTO*.

[72] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *ICST*.

[73] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *S&P*.

[74] K. Jiang, Y. Bao, S. Wang, Z. Liu, and T. Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In *CCS*.

[75] Marc Joye and Christophe Tymen. 2001. Protections against Differential Analysis for Elliptic Curve Cryptography. In *CHES*.

[76] M. Joye and S-M Yen. 2002. The Montgomery Powering Ladder. In *CHES*.

[77] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *CHES*.

[78] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS*.

[79] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security*.

[80] Y. Kim, R. Daly, J. S. Kim, C. Fallin, J-H Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*.

[81] J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM (1976)* (1976).

[82] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.

[83] P. C. Kocher, J. Jaffe, and B. Jun. 1999. Differential Power Analysis. In *CRYPTO*.
[84] Boris Köpf and Heiko Mantel. 2007. Transformational typing and unification for automatically correcting insecure programs. *Int. J. Inf. Sec.* (2007).
[85] Adam Langley. 2010. Ctgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html.
[86] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*.
[87] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency Throttling Side-Channel Attack. In *CCS*.
[88] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.
[89] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2022. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* 54, 6 (2022), 122:1–122:37.
[90] C-K Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. Geoffrey Lowney, S. Wallace, V. Janapa Reddi, and K. M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. (2005).
[91] Heiko Mantel, Alexandra Weber, and Boris Köpf. 2017. A Systematic Study of Cache Side Channels Across AES Implementations. In *ESSoS*.
[92] Thomas S. Messerges. 2000. Securing the AES Finalists Against Power Analysis Attacks. In *FSE*.
[93] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*.
[94] M. Nemec, D. Klinec, P. Svenda, P. Sekan, and V. Matyas. 2017. Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans. In *ACSAC*.
[95] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: differential fuzzing for side-channel analysis. In *ICSE*.
[96] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.
[97] P. Pessl, L. Groot Bruinderink, and Y. Yarom. 2017. To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In *CCS*.
[98] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*.
[99] Thomas Pornin. [n. d.]. BearSSL. https://bearssl.org/.
[100] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *S&P*.
[101] I. Puddu, M. Schneider, M. Haller, and S. Capkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *USENIX Security*.
[102] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. 2021. Systematic Analysis of Randomization-based Protected Cache Architectures. In *S&P*.
[103] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*.
[104] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security*.
[105] Josyula R. Rao and Pankaj Rohatgi. 2001. EMpowering Side-Channel Attacks. *IACR Cryptol. ePrint Arch.* (2001). http://eprint.iacr.org/2001/037
[106] O. Reparaz, J. Balasch, and I. Verbauwhede. 2017. Dude, is my code constant time?. In *DATE*.
[107] B. Rodrigues, F. Magno Quintão Pereira, and D. F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *CC*.
[108] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. 2019. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *S&P*.
[109] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. 2018. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *CCS*.
[110] Alexander Schaub. 2020. *Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations.* Ph. D. Dissertation. Institut Polytechnique de Paris. https://theses.hal.science/tel-03205242
[111] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.
[112] Martin Schwarzl, Erik Kraft, and Daniel Gruss. 2023. Layered Binary Templating: Efficient Detection of Compiler- and Linker-introduced Leakage. In *ACNS*.
[113] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX ATC*.
[114] Y-j Shin, H. Chan Kim, D. Kwon, J-H Jeong, and J. Hur. 2018. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS*.
[115] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *EuroS&P*.
[116] Luigi Soares and Fernando Magno Quintão Pereira. 2021. Memory-Safe Elimination of Side Channels. In *CGO*. IEEE, 200–210.
[117] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation. In *ASE*.
[118] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. Hardw. Syst. Secur.* 3, 3 (2019), 219–234.
[119] M. Tibouchi and A. Wallet. 2021. One Bit is All It Takes: A Devastating Timing Attack on BLISS's Non-Constant Time Sign Flips. *J. Math. Cryptol.* (2021).
[120] N. Tuveri, S. ul Hassan, C. Pereida García, and B. B. Brumley. 2018. Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study. In *ACSAC*.
[121] Rei Ueno and Naofumi Homma. 2023. How Secure Is Exponent-blinded RSA-CRT with Sliding Window Exponentiation? *TCHES* (2023).
[122] S. ul Hassan, I. Gridin, I. M. Delgado-Lozano, C. Pereida García, J-J Chi-Domínguez, A. Cabrera Aldaya, and B. B. Brumley. 2020. Déjà Vu: Side-Channel Analysis of Mozilla's NSS. In *CCS*.
[123] Jo Van Bulck. 2020. *Microarchitectural Side-channel Attacks for Privileged Software Adversaries.* Ph. D. Dissertation. KU Leuven.
[124] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*.
[125] Mathy Vanhoef and Eyal Ronen. 2020. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. In *S&P*.
[126] 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*.
[127] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security*.
[128] W. Wang, Y. Zhang, and Z. Lin. 2019. Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries. In *RAID*.
[129] Y. Wang, R. Paccagnella, E. Tang He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *USENIX Security*.
[130] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. 2020. Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In *USENIX Security*.
[131] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. 2018. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *AsiaCCS*.
[132] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security*.
[133] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC*.
[134] J. Wichelmann, F. Sieck, A. Pätschke, and T. Eisenbarth. 2022. Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. In *CCS*.
[135] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *ISSTA*.
[136] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS*.
[137] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*.
[138] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryp. ePrint* (2014).
[139] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[140] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. Butler, and D. J. Tian. 2022. ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs. *Transactions on Dependable and Secure Computing* (2022).
[141] Y. Yuan, Z. Liu, and S. Wang. 2023. CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software. In *USENIX Security*.
[142] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Automated Side Channel Analysis of Media Software with Manifold Learning. In *USENIX Security*.
[143] Tao Zhang, Timothy Lesch, Kenneth Koltermann, and Dmitry Evtyushkin. 2022. STBPU: A Reasonably Secure Branch Prediction Unit. In *DSN*.
[144] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *CCS*.