
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.

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

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

---

1https://github.com/ageimer/sok-detection/
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 cryptographic 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.

<table>
<thead>
<tr>
<th>Ref</th>
<th>Year</th>
<th>Tool</th>
<th>Type</th>
<th>Methods</th>
<th>Scal.</th>
<th>Policy</th>
<th>Sound</th>
<th>Input</th>
<th>L</th>
<th>W</th>
<th>E</th>
<th>B</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>[85]</td>
<td>2010</td>
<td>ct-grind</td>
<td>Dynamic</td>
<td>Tainting</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[15]</td>
<td>2013</td>
<td>Almeida et al.</td>
<td>Static</td>
<td>Deductive verification</td>
<td>CT</td>
<td></td>
<td></td>
<td>C source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[55]</td>
<td>2013</td>
<td>CacheAuditor</td>
<td>Static</td>
<td>Abstract interpretation</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[32]</td>
<td>2014</td>
<td>VIRTUALCERT</td>
<td>Static</td>
<td>Type system</td>
<td>CT</td>
<td></td>
<td></td>
<td>C source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[70]</td>
<td>2015</td>
<td>Cache Templates</td>
<td>Dynamic</td>
<td>Statistical tests</td>
<td>CO</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[13]</td>
<td>2016</td>
<td>ct-verif</td>
<td>Static</td>
<td>Logical verification</td>
<td>CT</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[107]</td>
<td>2016</td>
<td>FlowTracker</td>
<td>Static</td>
<td>Type system</td>
<td>CT</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[56]</td>
<td>2017</td>
<td>CacheAudit2</td>
<td>Static</td>
<td>Abstract interpretation</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[28]</td>
<td>2017</td>
<td>Blazy et al.</td>
<td>Static</td>
<td>Abstract interpretation</td>
<td>CT</td>
<td></td>
<td></td>
<td>C source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[17]</td>
<td>2017</td>
<td>Blazer</td>
<td>Static</td>
<td>Decomposition</td>
<td>CR</td>
<td></td>
<td></td>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[48]</td>
<td>2017</td>
<td>Thermis</td>
<td>Static</td>
<td>Logical verification</td>
<td>CR</td>
<td></td>
<td></td>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[127]</td>
<td>2017</td>
<td>CacheD</td>
<td>Dynamic</td>
<td>DSE</td>
<td>CO</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[136]</td>
<td>2017</td>
<td>STACCO</td>
<td>Dynamic</td>
<td>Trace diff</td>
<td>CR</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[106]</td>
<td>2017</td>
<td>ducket</td>
<td>Dynamic</td>
<td>Statistical tests</td>
<td>CC</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[117]</td>
<td>2018</td>
<td>CANAL</td>
<td>Static</td>
<td>SE</td>
<td>CO</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[47]</td>
<td>2018</td>
<td>CacheFix</td>
<td>Static</td>
<td>SE</td>
<td>CO</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[34]</td>
<td>2018</td>
<td>CoCo-Channel</td>
<td>Static</td>
<td>SE, tainting</td>
<td>CR</td>
<td></td>
<td></td>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[19]</td>
<td>2018</td>
<td>SideTrail</td>
<td>Static</td>
<td>Logical verification</td>
<td>CR</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[114]</td>
<td>2018</td>
<td>Shin et al.</td>
<td>Dynamic</td>
<td>Statistical tests</td>
<td>CO</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[132]</td>
<td>2018</td>
<td>DATA</td>
<td>Dynamic</td>
<td>Statistical tests</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[133]</td>
<td>2018</td>
<td>MicroWalk</td>
<td>Dynamic</td>
<td>MIA</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[110]</td>
<td>2019</td>
<td>STAnalyzer</td>
<td>Static</td>
<td>Abstract interpretation</td>
<td>CT</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[95]</td>
<td>2019</td>
<td>DnFuzz</td>
<td>Dynamic</td>
<td>Fuzzing</td>
<td>CR</td>
<td></td>
<td></td>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[126]</td>
<td>2019</td>
<td>CacheS</td>
<td>Static</td>
<td>Abstract interpretation, SE</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[35]</td>
<td>2019</td>
<td>CaSym</td>
<td>Static</td>
<td>SE</td>
<td>CO</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[54]</td>
<td>2020</td>
<td>Pitchfork</td>
<td>Static</td>
<td>SE, tainting</td>
<td>CT</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[66]</td>
<td>2020</td>
<td>ABSynthe</td>
<td>Dynamic</td>
<td>Genetic algorithm, RNN</td>
<td>CR</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[72]</td>
<td>2020</td>
<td>ct-fuzz</td>
<td>Dynamic</td>
<td>Fuzzing</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[51]</td>
<td>2020</td>
<td>Bnnsrc/Rnx</td>
<td>Static</td>
<td>SE</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[20]</td>
<td>2021</td>
<td>Abacus</td>
<td>Dynamic</td>
<td>DSE</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[74]</td>
<td>2022</td>
<td>CaType</td>
<td>Dynamic</td>
<td>Type system</td>
<td>CO</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[134]</td>
<td>2022</td>
<td>MicroWalk-CI</td>
<td>Dynamic</td>
<td>MIA</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary, JS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[140]</td>
<td>2022</td>
<td>ENCIDER</td>
<td>Static</td>
<td>SE</td>
<td>CT</td>
<td></td>
<td></td>
<td>LLVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[141]</td>
<td>2023</td>
<td>CacheQL</td>
<td>Dynamic</td>
<td>MIA, NN</td>
<td>CT</td>
<td></td>
<td></td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 prescribes 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-resource 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 BOOJIE 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. Executed execution paths are associated with a logical formula: the conjunction of conditions 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 multiple 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 oracle 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. DrFuzz [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] re-purposes 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. We have 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 Abacys’ 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 in which known-vulnerable functions (e.g., vulnerable functions are used in new contexts) 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]

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.

5.1.2 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, and 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.

5.2 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 codebase. 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 et-grind [85]. Together, they are representative of the diversity of methods in 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 counterexample 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.

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.
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 experiments (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 v.1.5 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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Binsec/Rel</th>
<th>Binsec/Rel2</th>
<th>Abacus</th>
<th>ctgrind</th>
<th>Microwalk</th>
<th>dudect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-CBC-bearssl (T)</td>
<td>20 ○ 0.05</td>
<td>0 36 ○ 0.02 0.10 36 ○ 3.65 36 ○ 0.16 36 ○ 1.39</td>
<td>100.31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-CBC-bearssl (BS)</td>
<td>20 ● – 16.77</td>
<td>0 ● – 0.31 0 ○ 10.69</td>
<td>0 ○ 0.17</td>
<td>0</td>
<td>1.55</td>
<td></td>
</tr>
<tr>
<td>AES-GCM-bearssl (T)</td>
<td>20 ○ 0.05</td>
<td>0 36 ○ 0.02 0.22 36 ○ 6.23</td>
<td>36 ○ 0.18</td>
<td>36 ○ 0.98</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>AES-GCM-bearssl (BS)</td>
<td>0 ● – 53.52</td>
<td>0 ● – 0.48 0 ○ 14.79</td>
<td>0 ○ 0.17</td>
<td>0</td>
<td>1.73</td>
<td></td>
</tr>
<tr>
<td>AES-CBC-mbedtls (T)</td>
<td>20 ○ 0.1</td>
<td>68 ○ 0.03 0.17 68 ○ 5.73</td>
<td>68 ○ 0.19</td>
<td>68 ○ 1.54</td>
<td>0.63</td>
<td></td>
</tr>
<tr>
<td>AES-GCM-mbedtls (T)</td>
<td>20 ○ 0.27</td>
<td>84 ○ 0.05 0.4 84 ○ 261.76</td>
<td>76 ○ 0.19</td>
<td>71 ○ 0.90</td>
<td>0.89</td>
<td></td>
</tr>
<tr>
<td>AES-CBC-openssl (EVP)</td>
<td>0 ● –</td>
<td>0 ○ 0.17 21.92</td>
<td>0 ○ 103.59</td>
<td>0 ○ 0.66</td>
<td>9 ○ 5.35</td>
<td></td>
</tr>
<tr>
<td>AES-GCM-openssl (EVP)</td>
<td>0 ● –</td>
<td>0 ○ 0.17 21.19</td>
<td>0 ○ 104.27</td>
<td>70 ○ 0.71</td>
<td>8 ○ 5.66</td>
<td></td>
</tr>
<tr>
<td>AES-CBC-openssl (T)</td>
<td>20 ○ 0.05</td>
<td>0 36 ○ 0.02 0.16 36 ○ 3.02</td>
<td>36 ○ 0.18</td>
<td>20 ○ 1.38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-CBC-openssl (VP)</td>
<td>0 τ ● –</td>
<td>0.06 0 ○ – 0.59</td>
<td>0 ○ 1.01</td>
<td>0 ○ 0.19</td>
<td>0 ○ 1.68</td>
<td></td>
</tr>
<tr>
<td>PolyChacha-bearssl (CT)</td>
<td>0 ● –</td>
<td>9.72 0 ● – 0.18 0 ○ 1.3</td>
<td>0 ○ 0.16</td>
<td>0 ○ 1.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PolyChacha-bearssl (SSE2)</td>
<td>0 ● –</td>
<td>0.05 0 ● – 0.11 0 ○ 1.17</td>
<td>0 ○ 0.15</td>
<td>0 ○ 1.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PolyChacha-mbedtls</td>
<td>0 ● –</td>
<td>18.62 0 ● – 0.49</td>
<td>0 ○ 5.02</td>
<td>0 ○ 0.18</td>
<td>0 ○ 2.95</td>
<td></td>
</tr>
<tr>
<td>PolyChacha-openssl (EVP)</td>
<td>0 ● –</td>
<td>0 ○ 0.17 21.55</td>
<td>0 ○ 100.19</td>
<td>0 ○ 0.67</td>
<td>8 ○ 5.41</td>
<td></td>
</tr>
<tr>
<td>Chacha20-openssl</td>
<td>0 ● –</td>
<td>0.77 0 ● – 0.09</td>
<td>0 ○ 3.77</td>
<td>0 ○ 0.15</td>
<td>0 ○ 1.36</td>
<td></td>
</tr>
<tr>
<td>Poly1305-openssl</td>
<td>0 ● –</td>
<td>0.26 0 ○ – 0.35</td>
<td>0 ○ 1.19</td>
<td>0 ○ 0.17</td>
<td>0 ○ 1.70</td>
<td></td>
</tr>
<tr>
<td>RSA-bearssl (OAEF)</td>
<td>0 ● –</td>
<td>2 ● 0.03</td>
<td>0 ○ 356.41</td>
<td>87 ○ 0.57</td>
<td>0 ○ 146.52</td>
<td></td>
</tr>
<tr>
<td>RSA-mbedtls (PKCS)</td>
<td>0 ○ 1.60</td>
<td>5 ○ 0.21</td>
<td>○ 2192.3</td>
<td>39 ○ 0.96</td>
<td>137 ○ 826.23</td>
<td></td>
</tr>
<tr>
<td>RSA-mbedtls (OAEF)</td>
<td>0 ○ 1.46</td>
<td>5 ○ 0.21</td>
<td>○ 2203.89</td>
<td>48 ○ 0.98</td>
<td>137 ○ 847.27</td>
<td></td>
</tr>
<tr>
<td>RSA-openssl (PKCS)</td>
<td>1 ○ 26.50</td>
<td>1 ○ 0.44</td>
<td>○ 551.72</td>
<td>321 ○ 1.32</td>
<td>46 ○ 52.06</td>
<td>618.73</td>
</tr>
<tr>
<td>RSA-openssl (OAEF)</td>
<td>1 ○ 29.02</td>
<td>1 ○ 0.46</td>
<td>○ 535.91</td>
<td>546 ○ 1.73</td>
<td>61 ○ 59.90</td>
<td>771.3</td>
</tr>
<tr>
<td>ECDSA-bearssl</td>
<td>2 ○ 1.82</td>
<td>2 ○ 0.84</td>
<td>○ 246.62</td>
<td>7 ○ 0.41</td>
<td>0 ○ 109.31</td>
<td></td>
</tr>
<tr>
<td>ECDSA-mbedtls</td>
<td>0 ● –</td>
<td>0.54 0 ○ – 0.19</td>
<td>0 ○ 467.11</td>
<td>50 ○ 0.81</td>
<td>132 ○ 314.26</td>
<td>224.30</td>
</tr>
<tr>
<td>ECDSA-openssl</td>
<td>0 ● –</td>
<td>2252.36 1 ○ 50.</td>
<td>0 ○ 202.72</td>
<td>14 ○ 1.22</td>
<td>28 ○ 13.76</td>
<td></td>
</tr>
<tr>
<td>EdDSA-openssl</td>
<td>0 ● –</td>
<td>26.36 0 ○ – 25.63</td>
<td>0 ○ 59.87</td>
<td>0 ○ 1.03</td>
<td>8 ○ 9.86</td>
<td></td>
</tr>
</tbody>
</table>

- ●: the tool crashed; τ: 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 (-runtime exceeds the limit). 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/Rel, 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 analysis’ 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 exploitable, as even a single bit of leakage or less can be exploited [18].

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 (nV) 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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Binsec/Rel2</th>
<th>Abacus</th>
<th>ctgrind</th>
<th>Microwalk</th>
<th>dudect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V #V S T (s)</td>
<td>V #V S T (s)</td>
<td>V #V S T (s)</td>
<td>V #V S T (s)</td>
<td>S T (s)</td>
</tr>
<tr>
<td>P256 sign (OpenSSL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wNAF multiplication</td>
<td>5</td>
<td>○</td>
<td>⚫</td>
<td>3</td>
<td>–</td>
</tr>
<tr>
<td>RSA valid. (MbedTLS)</td>
<td>5</td>
<td>○</td>
<td>●</td>
<td>3</td>
<td>–</td>
</tr>
<tr>
<td>GCD</td>
<td>3</td>
<td>○</td>
<td>●</td>
<td>3</td>
<td>–</td>
</tr>
<tr>
<td>modular inversion</td>
<td>2</td>
<td>○</td>
<td>●</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>RSA keygen (OpenSSL)</td>
<td>3</td>
<td>○</td>
<td>●</td>
<td>3</td>
<td>–</td>
</tr>
<tr>
<td>GCD</td>
<td>2</td>
<td>○</td>
<td>●</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>modular exponentiation</td>
<td>1</td>
<td>○</td>
<td>●</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>ECDH decryption (Libgcrypt)</td>
<td>0</td>
<td>○</td>
<td>●</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>modular reduction</td>
<td>0</td>
<td>○</td>
<td>●</td>
<td>0</td>
<td>–</td>
</tr>
</tbody>
</table>

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. Conversely, 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 regard. 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 etgrind. 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


