"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks

Jan Jancar*, Marcel Fourné†, Daniel De Almeida Braga‡, Mohamed Sabt‡,
Peter Schwabe†§, Gilles Barthe†¶, Pierre-Alain Fouque‡ and Yasemin Acar‖†
*Masaryk University, Brno, Czech Republic †MPI-SP, Bochum, Germany
‡Rennes University, CNRS, IRISA, Rennes, France §Radboud University, Nijmegen, The Netherlands
¶IMDEA Software Institute, Madrid, Spain ‖The George Washington University, Washington D.C., USA

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833713

***Abstract*—Timing attacks are among the most devastating side-channel attacks, allowing remote attackers to retrieve secret material, including cryptographic keys, with relative ease. In principle, "these attacks are not that hard to mitigate": the basic intuition, captured by the *constant-time* criterion, is that control-flow and memory accesses should be independent from secrets. Furthermore, there is a broad range of tools for automatically checking adherence to this intuition. Yet, these attacks still plague popular cryptographic libraries twenty-five years after their discovery, reflecting a dangerous gap between academic research and cryptographic engineering. This gap can potentially undermine the emerging shift towards high-assurance, formally verified cryptographic libraries. However, the causes for this gap remain uninvestigated.**

To understand the causes of this gap, we conducted a survey with 44 developers of 27 prominent open-source cryptographic libraries. The goal of the survey was to analyze if and how the developers ensure that their code executes in constant time. Our main findings are that developers are aware of timing attacks and of their potentially dramatic consequences and yet often prioritize other issues over the perceived huge investment of time and resources currently needed to make their code resistant to timing attacks. Based on the survey, we identify several shortcomings in existing analysis tools for constant-time, and issue recommendations that can make writing constant-time libraries less difficult. Our recommendations can inform future development of analysis tools, security-aware compilers, and cryptographic libraries, not only for constant-timeness, but in the broader context of side-channel attacks, in particular for micro-architectural side-channel attacks, which are a younger topic and too recent as focus for this survey.

***Index Terms*—constant-time, timing attacks, cryptographic library, survey, developer survey, expert survey, usable security, human factors, cryptography**

I. Introduction

Cryptographic protocols, such as TLS (Transport Layer Security), are the backbone of computer security, and are used at scale for securing the Internet, the Cloud, and many other applications. Quite strikingly, the deployment of these protocols rests on a small number of open-source libraries, developed by a rather small group of outstanding developers. These developers have a unique set of skills that are needed for writing efficient, correct, and secure implementations of (often sophisticated) cryptographic routines; in particular, they combine an excellent knowledge of cryptography and of computer architectures and a deep understanding of low-level programming. Unfortunately, in spite of developers' skills and experience, new and sometimes far-reaching vulnerabilities and attacks are regularly discovered in major open-source cryptographic libraries. One class of vulnerabilities are *timing attacks*, which let an attacker retrieve secret material, such as cryptographic keys, *"by carefully measuring the amount of time required to perform private key operations"*. Although timing attacks were first described by Kocher in 1996 [1], they continue to plague implementations of cryptographic libraries. There are multiple aspects that make timing attacks *special* in comparison to other side-channel attacks such as power-analysis or EM attacks. First, they can be carried out *remotely*, both in the sense of running code in parallel to the victim code without the need of local access to the target computer, but also in the sense of only interacting with a server over the network and measuring network timings [2] or over the Cloud [3]. As a consequence, unlike many other side-channel attacks, timing attacks cannot be prevented by restricting physical access to the target machine. Second, timing attacks do not leave traces on the victim's machine beyond possibly suspicious access logs, and we do not know at all to what extent they are being carried out in the real world, for example by government agencies: victims are not able to reliably detect that they are under attack and the attackers will never tell.

At the same time, and most importantly for this paper, we know how to systematically protect against timing attacks. The basic idea of such systematic countermeasures was already described by Kocher in 1996 [1]: we need to ensure that all code takes time independent of secret data. It is important here to not just consider the total time taken by some cryptographic computation, but make sure that this property holds for each instruction. This paradigm is known as *constant-time*[1] cryptography and is usually achieved by ensuring that

- there is no data flow from secrets into branch conditions;

[1]The term constant-time, often referred as CT, is a bit of a misnomer, as it does not refer to CPU execution time but rather to a structural property of programs. However, it is well-established in the cryptography community.

DOI 10.1109/SP46214.2022.00044

- addresses used for memory access do not depend on secret data; and
- no secret-dependent data is used as input to variable-time arithmetic instructions (such as, e.g., `DIV` on most Intel processors or `SMULL`/`UMULL` on ARM Cortex-M3).

Constant-timeness is no panacea, and the above rules may not be sufficient on some micro-architectures or in the presence of speculative execution, but essentially all timing-attack vulnerabilities found so far in cryptographic libraries could have been avoided by following these rules. For this reason, the notion of constant-time has grown in importance in standardization processes and recent cryptographic competitions. For instance, in the context of the ongoing Post-Quantum Cryptography Standardization project, the National Institute of Standards and Technology have stated in their Call for Papers [4]:

> *"Schemes that can be made resistant to side-channel attack at minimal cost are more desirable than those whose performance is severely hampered by any attempt to resist side-channel attacks. We further note that optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not."*

Protection against side-channel attacks, including timing attacks, is also routinely included as a requirement for Common Criteria certification as well as a part of the newly approved FIPS 140-3 certification scheme [5].

Programming highly optimized code that is also constant-time can be very challenging. However, we know how to verify that programs are constant-time. This was first demonstrated by Adam Langley's ctgrind [6], developed in 2010, the first tool to support analysis of constant-timeness. A decade later, there are now more than 30 tools for checking that code satisfies constant-timeness or is resistant against side-channels [7], [8]. These tools differ in their goals, achievements, and status. Yet, they collectively demonstrate that automated analysis of constant-time programs is feasible; for instance, a 2019 review [8] lists automatic verification of constant-time real-world code as one achievement of computer-aided cryptography, an emerging field that develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography.

Based on this state of affairs, one would expect that timing leaks in cryptographic software have been systematically eliminated, and timing attacks are a thing of the past. Unfortunately, this is far from true, so in this paper we set out to answer the question:

Why is today's cryptographic software not free of timing-attack vulnerabilities?

More specifically, to understand how real-world cryptographic library developers think about timing attacks and the constant-time property, as well as constant-time verification tools, we conducted a mixed-methods online survey with 44 developers of 27 popular cryptographic libraries / primitives[2]. Through this survey, we track down the origin of the persistence of timing attacks by addressing multiple sub-questions:

[2]We refer to both as "libraries" for readability.

RQ1: (a) Are timing attacks part of threat models of libraries/primitives? (b) Do libraries and primitives claim resistance against timing attacks?
RQ2: (a) How do libraries/primitives protect against timing attacks? (b) Are libraries and primitives being verified/tested for constant-timeness? (c) How often/when is this done?
RQ3: (a) What is the state of awareness of tools that can verify constant-timeness? (b) What are the experiences with the tools?
RQ4: Are participants inclined to hypothetically use formal-analysis-based, dynamic instrumentation, or runtime statistical test tools, based on tool use requirements and guarantees?
RQ5: What would developers want from constant-time verification tools?

We find that, while all 44 participants are aware of timing attacks, not all cryptographic libraries have verified/tested resistance against timing attacks. Reasons for this include varying threat models, a lack of awareness of tooling that supports testing/verification, lack of availability, as well as a perceived significant effort in using those tools (see Figure 1). We expose these reasons, and provide recommendations to tool developers, cryptographic libraries developers, compiler writers, and standardization bodies to overcome the main obstacles towards a more systematic protection against timing attacks. We also briefly discuss how these recommendations extend to closely related lines of research, including tools for protecting against Spectre-style attacks [9].

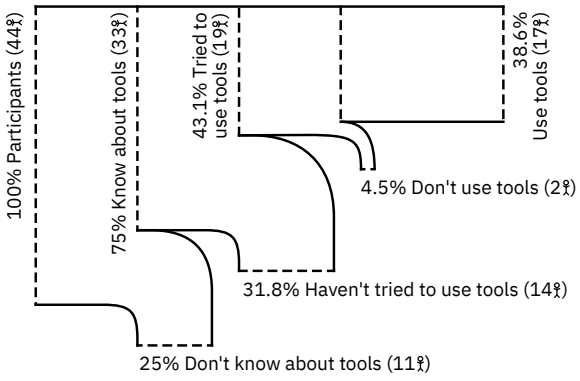

Fig. 1. Leaky pipeline of developers' knowledge and use of tools for testing or verifying constant-timeness.

II. Background & Related Work

A. Attacks

In 1996, Kocher introduced the concept of *timing attacks* as a means to attack cryptographic implementations *"by carefully measuring the amount of time required to perform private key operations"* [1]. He described successful timing attacks against implementations of various building blocks commonly used in asymmetric cryptography like modular exponentiation and Montgomery reduction against RSA and DSS. Since this seminal paper, timing attacks have been further refined and continued to plague implementations of both asymmetric and

symmetric cryptography. Successful timing attacks are way too numerous to list all, so we focus on a few relevant examples.

In 2002, Tsunoo et al. [10], [11] were the first to present attacks exploiting cache timing to break symmetric cryptography (MISTY1 and DES); they also mentioned a cache-timing attack against AES. Details of cache-timing attacks against AES were first presented in independent concurrent work by Bernstein [12] and by Osvik, Shamir, and Tromer [13]. In 2003, D. Brumley and Boneh showed that timing attacks can be mounted remotely by measuring timing variations in response times of SSL servers over the network [2]. Canvel et al. showed in 2003 how to recover passwords in SSL/TLS channels using padding oracle attacks [14]. In 2011, B. Brumley and Tuveri showed that such remote attacks are still possible [15]; i.e., that the underlying weaknesses in the OpenSSL library had not been suitably fixed. SSL libraries continued to be the target of timing attacks; examples include the "Lucky 13" attack by AlFardan and Paterson, which exploits timing variation in the processing of padding in the CBC mode of operation in multiple common SSL/TLS libraries [16] similar in principle to the paper by Canvel et al. [14]. In 2015, Albrecht and Paterson presented a variant of the attack targeting Amazon's s2n implementation of TLS [17]. In 2016, Yarom, Genkin, and Heninger presented the "CacheBleed" attack, which showed that the "scatter-gather" implementation technique recommended by Intel [18] and implemented in OpenSSL as cache-timing attack countermeasure, is insufficient to thwart attacks [19]. In the same year, Kaufmann et al. showed that even carefully implemented C code may be translated to binaries that are vulnerable to timing attacks [20].

We conclude this paragraph with a few attacks related to certification and standardization. Certification schemes such as Common Criteria often require certified products to have countermeasures to a range of side-channel attacks, including timing attacks. However, certified hardware did not avoid being a target of timing attacks, as shown by the recent Minerva group of vulnerabilities in ECDSA implementations, including a Common Criteria certified smartcard [21]. In recent years, various timing attacks were proposed against implementations of post-quantum cryptography (PQC) including attacks against the BLISS signature scheme used in the strongSwan IPsec implementation [22]–[24] and attacks against candidates in NIST's PQC standardization effort [25]–[27].

Despite all these academic timing attacks, their practical exploitability is often questioned by practitioners. Security Audit companies try to catch timing vulnerabilities in software [28]. However, they make the following statements:

> *"Even though there is basic awareness of timing side-channel attacks in the community, they often go unnoticed or are flagged during code audits without a true understanding of their exploitability in practice."*

B. Tools included in the survey

We provide a brief overview of the tools considered in our survey. We classify tools according to the broad approach they use: runtime statistical tests, dynamic-instrumentation based, or formal-analysis based. Our approach as well as our choice of included tools is based on an earlier paper [8], but amended with tools some authors know to be in current use.

Broadly speaking, statistical test tools [29] compute the execution time of a large number of runs of the target program and verify whether secret data influences the execution time. These tools are generally easy to install and run, even at scale, and operate on executable code, ruling out the possibility of compiler-induced violations of the constant-time policy. However, they only provide weak, informal guarantees.

In contrast, dynamic instrumentation based tools [6], [30]–[41] instrument programs to track how information flows during (concrete or symbolic) execution of programs. They are generally reasonably easy to install and to use, even at scale, and can be implemented at source, intermediate, or assembly levels, and provide formal guarantees. However, as with all tools based on dynamic techniques, these guarantees are generally limited; for instance, dynamic analysis of loops may be unsound, i.e., miss constant-time violations.

Finally, formal-analysis-based tools [42]–[52] provide strong guarantees that programs do not violate constant-timeness; in addition, some tools are precise, in that they only reject programs that violate constant-timeness. Their other criterion is soundness, which ensures the absence of constant-time violations. However, these tools are often implemented at source or intermediate levels, frequently require user interaction, and are sometimes hard to install or use at scale.

Table I presents some key tools and summarizes their main characteristics. Since our focus is not an in-depth technical comparison of the features of the tools, we deliberately keep descriptions simple, and only consider their target and whether they provide some formal guarantees (No, Partial, Yes, Other). For the cognizant, "Partial guarantees" cover tools that perform dynamic analysis, whereas "Guarantees" cover tools that are sound and detect all constant-time violations; in particular, our classification does not reflect if tools are precise. Even for such coarse criteria, classification is sometimes challenging so we err on the generous side. Finally, we tag tools as "Other" if they establish another property than constant-time; comparing these properties with constant-time is often tricky, so we choose not to qualify the difference.

While the CoCo-Channel authors wrote [33]: *"We also evaluate CoCo-Channel against two recent tools for detecting side-channel vulnerabilities in Java applications, Blazer and Themis. Neither are publicly available[...]"*, their tool was not found by us either.

We do not claim our list to be comprehensive, especially in this currently active field of research. In particular, we did not ask about Constantine [54], Pitchfork-angr [55], Cachefix [56], and ENCoVer [57], just to name a few.

C. Libraries included in the survey

Cryptographic libraries have diverse threat models, but with their usual use in protocols like TLS and connected applications often running on shared hardware, resistance against timing attacks is an important property. In our survey,

Tool	Target	Techn.	Guarantees
ABPV13 [42]	C	Formal	●
Binsec/Rel [30]	Binary	Symbolic	◐
Blazer [43]	Java	Formal	●
BPT17 [31]	C	Symbolic	◐
CacheAudit [44]	Binary	Formal	■
CacheD [32]	Trace	Symbolic	○
COCO-CHANNEL [33]	Java	Symbolic	●
ctgrind [6]	Binary	Dynamic	◐
ct-fuzz [34]	LLVM	Dynamic	○
ct-verif [45]	LLVM	Formal	●
CT-WASM [46]	WASM	Formal†	●
DATA [35], [36]	Binary	Dynamic	◐
dudect [29]	Binary	Statistics	○
FaCT [47]	DSL	Formal†	●
FlowTracker [48]	LLVM	Formal	●
haybale-pitchfork [37]	LLVM	Symbolic	◐
KMO12 [49]	Binary	Formal	■
MemSan [38]	LLVM	Dynamic	◐
MicroWalk [39]	Binary	Dynamic	◐
SC-Eliminator [53]	LLVM	Formal†	●
SideTrail [50]	LLVM	Formal	■
Themis [51]	Java	Formal	●
timecop [40]	Binary	Dynamic	◐
tis-ct [41]	C	Symbolic	◐
VirtualCert [52]	x86	Formal	●

Targets: LLVM - intermediate representation, DSL - domain-specific language, WASM - Web Assembly
Technique: † - also performs code transformation/synthesis
Guarantees: ● - sound, ◐ - sound with restrictions, ○ - no guarantee, ■ - other property

TABLE I
CLASSIFICATION OF TOOLS INCLUDED IN THE SURVEY.

we invited developers of all widely used TLS libraries and other smaller but popular libraries and relevant primitives. We focused on libraries implemented in C/C++ as it is the target language of most tools and the most used language for cryptographic libraries. However, we included some libraries implemented in Java, Rust and Python if some tools can analyse them or they contain parts implemented in C.

Our choice of libraries is underpinned not only by our knowledge of them but also by quantitative data of user and developer numbers. We included some newer primitives not (yet) fulfilling this criterion to complement the answers given by the first group. Nemec et al. [58] gave numbers for OpenSSL: *"The prevalence of OpenSSL reaches almost 85% within the current Alexa top 1M domains and more than 96% for client-side SSH keys as used by GitHub users."* We only included libraries with an open development model to allow us to get data for our recruiting choice.

Table II contains a list of libraries included in the survey and whether at least one of their developers participated in our survey. The table also lists the actions that the libraries perform in their Continuous Integration (CI) pipelines. We draw this information from documentation and the public CI pipelines of the libraries. One author extracted this, a second author double-checked, with disagreements discussed and resolved.

D. Additional Related Work

Having already discussed timing attacks and tools for constant-time analysis, we briefly cover other related work.

Library	Particip.	Continuous integration			
		Build	Test	Fuzz‡	CT test
OpenSSL	✓	✓	✓	✓	✗
LibreSSL	✓	✓	✓	✓	✗
BoringSSL	✓	✓	✓	✓	✓
BearSSL	✓	✓	✓	✗	✗
Botan	✓	✓	✓	✓	✓
Crypto++		✓	✓	✗	✗
wolfSSL	✓	✓	✓	✓	✗
mbedTLS	✓	✓	✓	✓	✓
Amazon s2n	✓	✓	✓	✓	✓
MatrixSSL			No public CI		
GnuTLS	✓	✓	✓	✓	✗
NSS	✓	✓	✓	✓	✗
libtomcrypt	✓	✓	✓	✗	✗
libgcrypt	✓		No public CI		
Nettle	✓	✓	✓	✓	✗
Microsoft SymCrypt	✓	✓	✓	✓	✗
Intel IPP crypto			No public CI		
cryptlib	✓		No public CI		
libsecp256k1	✓	✓	✓	✓	✓
NaCl	✓		No public CI		
libsodium	✓	✓	✓	✓	✗
monocypher	✓	✓	✓	✓	✗
BouncyCastle*	✓	✓	✓	✗	✗
OpenJDK		✓	✓	✗	✗
dalek-cryptography†		✓	✓	✗	✗
ring†		✓	✓	✓	✗
RustCrypto†	✓	✓	✓	✗	✗
rustls†	✓	✓	✓	✓	✗
python-ecdsa	✓	✓	✓	✗	✗
micro-ecc			No public CI		
tiny-AES-c	✓	✓	✓	✗	✗
PQCrypto-SIDH	✓	✓	✓	✗	✓
csidh	✓		No public CI		
constant-csidh-c-implementation	✓		No public CI		
ARMv8-CSIDH			No public CI		
SPHINCS+		✓	✓	✗	✗
Total = 36	27 (75%)	27 (75%)	27 (75%)	16 (44%)	6 (17%)

* Java
† Rust
‡ Includes being fuzzed by OSS-Fuzz or cryptofuzz.

TABLE II
LIBRARIES AND PRIMITIVES INCLUDED AND THE ACTIONS THEY PERFORM IN THEIR PUBLIC CONTINUOUS INTEGRATION PIPELINES.

a) Foundations of constant-time programming: Constant-time programming is supported by rigorous foundations. These foundations typically establish that programs are protected against passive adversaries that observe program execution. However, Barthe et al. [52] show that constant-time programs are protected against system-level adversaries that control the cache (in prescribed ways) and the scheduler. Recently, these foundations have been extended to reflect micro-architectural attacks [59]–[62]. In parallel, a large number of tools are being developed to prove that programs are speculative-constant-time, a strengthening of the constant-time property which offers protection against Spectre [9] attacks. We expect that many of the takeaways of our work are applicable to this novel direction of work.

b) High-assurance cryptography: High-assurance cryptography is an emerging area that aims to build efficient implementations that achieve functional correctness, constant-

timeness, and security. High-assurance cryptography has already achieved notable successes [8]. The most relevant success in the context of this work is the EverCrypt library [63], [64], which has been deployed in multiple real-world systems, notably Mozilla Firefox and Wireguard VPN. The EverCrypt library is formally verified for constant-timeness (and functional correctness). However, the library is conceived as drop-in replacements for existing implementations, and despite relying on an advanced infrastructure built around the F* programming language, this work does not explicitly target open-source cryptographic library developers as potential users of their infrastructure. Other projects that enforce constant-time by default, such as Jasmin [65], [66] or FaCT [47], target open source cryptographic library developers more explicitly, but rely on domain-specific languages, which may hinder their broad adoption. In contrast, we focus on tools that do not impose a specific programming framework for developers.

c) Human factor research: Researchers have tried to answer the question of why cryptographic advances do not necessarily reach users. In a 2017 study, Acar et al. find that bad cryptographic library usability contributes to misuse, and therefore insecure code [67]. Krueger et al. developed and built upon a wizard to create secure code snippets for cryptographic use cases [68], [69]. Unlike these prior studies that investigate users of cryptographic libraries, we study the developers of cryptographic libraries, their threat models and decisions as they relate to *timing attacks*.

Haney et al. investigate the mindsets of those who develop cryptographic software, finding that company culture and security mindsets influence each other positively, but also that some cryptographic product developers do not adhere to software engineering best practices (e.g., they write their own cryptographic code) [70]. We expand on this research by surveying open-source cryptographic library developers with respect to their decisions and threat models as they relate to side-channel attacks.

In the setting of constant-time programming, Cauligi et al. [47] carry a study with over 100 UCSD students to understand the benefits of FaCT, a domain-specific framework that enforces constant-time at compile-time, with respect to constant-time programming in C. They find that tool support for constant-time programming is helpful. We expand on their study by surveying open-source cryptographic library developers and considering a large set of tools.

Very recently, there have been calls to make formal verification accessible to developers: Reid et al. suggest "meeting developers where they are" and integrating formal verification functionality in tools and workflows that developers are already using [71]. To our knowledge, ours is the first survey that empirically assesses cryptographic library developers' experiences with formal verification tools.

III. Methodology

In this section, we provide details on the procedure and structure of the survey we conducted with 44 developers of popular cryptographic libraries and primitives. We describe the coding process for qualitative data, as well as the approach for statistical analyses for quantitative results. We explain our data collection and ethical considerations, and discuss the limitations of this work.

A. Study Procedure

We asked 201 representatives of popular cryptographic libraries or primitives to participate in our survey. The recruited developers reside in different time zones and each may have different time constraints. As we were mainly interested in qualitative insights, based on the small number of qualifying individuals and our past experiences with low opt-in rates when attempting to recruit high-level open source developers into interview studies, we opted for a survey with free-text answers.

a) Questionnaire Development: We used our research questions as the basis for our questionnaire development, but we also let our experience with the development of cryptographic libraries, constant-time verification tools (both as authors as well as users), and conducting developer surveys influence the design. Our group of authors consists of one human factors researcher and experts from cryptographic engineering, side-channel attacks, and constant-time tool developers. The human factors researcher introduced and facilitated the use of human factors research methodology to answer experts' research questions posited in this paper. In particular, the human factor researcher explained methods when appropriate, facilitated many discussions and helped the team to develop the survey, pilot it, gather feedback, and evaluate the results. While iterating over the questionnaire, we also collected feedback and input from members of the cryptographic library development community.

b) Pre-Testing: Following the principle of cognitive interviews [72], we walked through the survey with three participants who belonged to our targeted population, and updated, expanded and clarified the survey accordingly.

c) Recruitment and Inclusion Criteria: We created a list of the most active contributors to libraries that implement cryptographic code, including those that implement cryptographic primitives. If a library had any formal committee for making technical decisions, we invited its members. The list of most active developers was extracted from source control by taking the developers with the largest amount of commits down to a cut-off point that was adjusted per library. Table II gives an overview of projects for which we invited participants. All authors then identified those contributors that belonged to their own personal or professional networks and invited those in a personalized email. All others were invited by a co-author who is active in the formal verification and cryptography community, for whom we assumed that they would be widely known and have the best chance of eliciting responses. All contributors were sent an invitation with a personalized link. We did not offer participants compensation, but offered them links to all the tools we mentioned in our survey, as well as the option to be informed about our results.

B. Survey Structure

The survey consisted of six sections (see Figure 2) detailed below. The full questionnaire can be found in Appendix A.

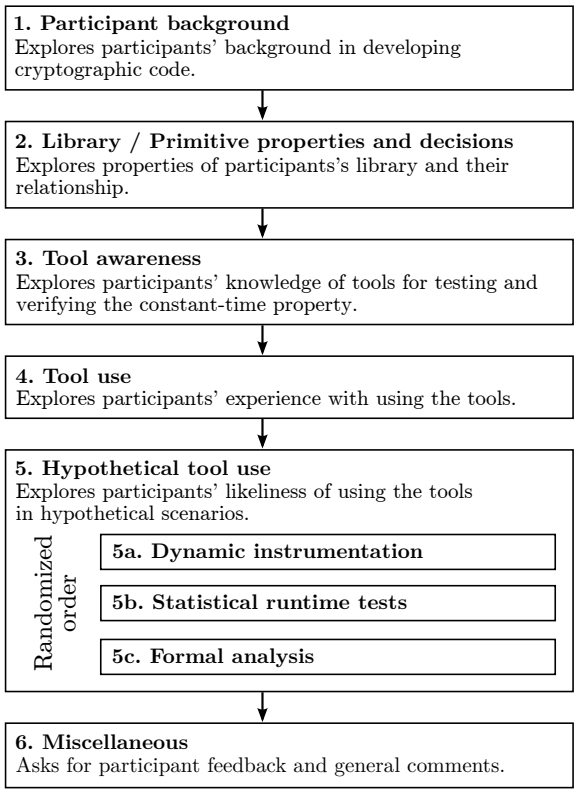

Fig. 2. Survey flow as shown to participants.

1. Participant background: We asked participants about their background in cryptography, their years of experience in developing cryptographic code, and their experience as a cryptographic library / primitive developer.

2. Library properties and decisions: We asked about participants' role in <library>'s development, how they are involved in design decisions for <library>. We asked about the intended use cases for <library>, <library>'s threat model with respect to side-channel attacks, whether they consider timing attacks a relevant threat for the intended use of <library> and its threat model, and asked for an explanation for their reasoning. We also asked whether and how <library> protects against timing attacks, and whether, how, and how often they test or verify resistance to timing attacks.

3. Tool awareness: We asked whether they were aware of tools that can test or verify resistance to timing attacks. We then listed 25 tools from Section II-B and asked them whether they were aware of them, and how they learned about them.

4. Tool use: We asked participants about their past experience, interactions, comprehension, and satisfaction with using tools to test/verify resistance to timing attacks, including challenges with using them.

5. Hypothetical tool use: We showed participants a description of properties that their code would have to fulfill in order to be able to use a group of tools and given a description of the guarantees the tools would give them, asking them about usage intentions and reasoning. The tools were grouped into dynamic instrumentation based, statistical test based, and formal analysis tools.

6. Miscellaneous: Finally, we asked about any comments on (resistance to) timing attacks, our survey, and whether they wanted us to inform them about our results.

C. Coding and Analysis

Those who engaged with participant responses came from different backgrounds, with different views, contributing to the multi-faceted evaluation. Three researchers familiar with constant-time verification and open-source cryptographic library development conducted the qualitative coding process, facilitated by one researcher with experience with human factors research with developers. We followed the process for thematic analysis [73]. The three coders familiarized themselves with all free-text answers, and annotated them. Based on these annotations, themes were developed, as well as a codebook. The codebook was developed inductively based on questions, and iteratively changed based on responses we extracted from the free-text answers; all codes were operationalized based on discussions within the team. The three coders then coded all responses with the codebook, iterating over the codebook until they were able to make unanimous decisions. The codebook codifies answers to free-text questions, as well as identifying misconceptions, concerns, and wishes. In some cases where documentation was available, and participant answers were incomplete or ambiguous, or when participants linked to documentation, coders supplemented their code assignment based on the documentation. Our coding process was only one step in the quest for our goal: identifying themes and answering our research questions. All codes were discussed, and eventually agreed upon by three coders[3]. In line with contemporary human factors research, we therefore omit inter-coder agreement calculations [74]. For the comparison of the likelihood of using certain tools with certain requirements in exchange for guarantees (Q5.1, Q6.1, Q7.1), we used Friedman's test with Durbin post-hoc tests [75] with Benjamini-Hochberg multiple testing corrections [76].

D. Data Collection and Ethics

While our survey was sent to open-source contributors without solicitation, we only emailed them up to twice. During and after the survey, they could opt-out of participation. We do not link participant names to results, nor participant demographics to libraries to keep responses as confidential as possible. We also do not link quotes to libraries or their developers, and report mostly aggregate data. Quotes are pseudonymized. Our study protocol and consent form were approved by our institution's data protection officer and ethics board and determined to be minimal risk. Participant names and email addresses were stored separately from study data, and only used for contacting participants.

[3]Our codebook is available at https://crocs.fi.muni.cz/public/papers/usablect_sp22.

E. Limitations

Like all surveys, our research suffers from multiple biases, including opt-in bias and self-reporting bias. However, we were pleasantly surprised that for 27 out of the 36 libraries we selected, we received at least one valid response. Participants may over-report desirable traits (like caring about side-channel attacks or protecting against them), and underplaying negative traits (like making decisions ad-hoc). However, their reporting generally tracked with official documents and our a priori knowledge about the libraries. The projects represent a selection, and are not representative of all cryptographic libraries. However, we took great care in inviting participants corresponding to a variety of prominent, widely used libraries as well as smaller but popular libraries and primitives, as assessed by multiple authors who work in this space.

F. Data cleaning & Presentation

We emailed 201[4] listed as most active contributors to 36 libraries/primitives, finding alternate emails for those emails that bounced. 2 emailed us to tell us that they did not think they could meaningfully contribute. In total, 71 started the survey. We removed all 25 incomplete responses. We removed two participants because they gave responses about a project of their own, instead of the library we asked them about. From here, we report results only for the 44 valid participants. For statistical testing and figures for hypothetical tool use, we report results for the 36 participants who gave answers for all three tool groups. We merged answers of participants talking about using a ctgrind-like approach but without the use of ctgrind itself (as it is no longer necessary as Valgrind can directly do this) into the ctgrind tool answers.

Participants spent an average of 32 minutes on the survey, and left rich free text comments. We generally received positive feedback and high interest in our work, and 35 asked to be sent our results, with 33 agreeing to be contacted for follow-up questions. Whenever we report results at the library level, we merge qualitative answers given by all participants corresponding to that library. Whenever the answers are additive, we add them together without reporting a conflict (e.g. when one developer tests a library in one way while another one tests it a different way, we report both). When the answers are claims of a level (e.g. resistance to timing attacks) we report the highest claimed. Otherwise, whenever we encounter conflicting opinions, we report on this conflict.

IV. Results

In this section, we answer our research questions based on the results of our survey. Between full awareness and low levels of protection against timing attacks, we identify reasons for (not) choosing to develop and verify constant-time code, including a lack of (easy-to-use) tooling, tradeoffs with competing tasks, understandable concerns and misconceptions about current tooling. We identify that participants would generally like the guarantees offered by tools, but fear negative experiences, code annotations and problems with scalability.

[4]From now on, we use the symbol to denote the participants.

A. Survey Participants

Library developers: We successfully recruit experienced cryptographic library developers, including the most active contributors and decision-makers. We ended up with 44 recruited via direct invitation. Of our participants, 4 were the only developer in their project, 9 were project leads, 11 were core developers, 19 were maintainers, 11 were committers, 3 were contributors without commit rights. These classes are non-exclusive self-reports. 40 said they were involved in the library decision processes, while only 4 were not involved.

Participants had strong backgrounds in cryptographic development, reporting a median of 10 years of experience (sd = 7.75), and qualitatively reporting strong engagement with various projects, for example reporting involvement in security certifications: *"I've worked on open source and closed source cryptography libraries, dealt with various Common Criteria EAL4+ products"* (P1). As for the participants' concrete background in cryptography, 17 reported an academic background, 15 took some classes on cryptography, 32 had on the job experience, 6 teach cryptography, for 15 cryptography is (also) a hobby, 27 have industry experience in cryptography.

Libraries: We ended up with participants from 27 prominent libraries, such as OpenSSL, BoringSSL, mbedTLS or libgcrypt. Participants gave or linked to descriptions of a broad range of use cases for cryptographic libraries. As intended platforms, 23 gave servers, 22 desktop, 14 embedded device (with OS, 32 bit), 4 mobile, and 1 micro-controller (no OS, 8/16 bit). For targets, 7 stated TLS, 12 protocols, 2 services, 1 cloud, 2 operating systems, 1 crypto-currency, and 2 corporate internal purposes. Libraries had varying decision-making processes: 9 made decisions by discussion, 2 by voting, 3 by consensus, and for 11, decisions were made by the project leads who had a final say.

B. Answering Research Questions

1) Threat models (RQ1a): Here, we answer the research question whether timing attacks are part of library developers' threat models (RQ1a). We found that *all participants were aware of timing attacks*. Generally, when a threat model is defined for a cryptographic library, it mostly includes timing attacks. However, strict and absolute adherence to constant-time code is most often not required. In practice, developers tend to distinguish vulnerabilities that are "easy" to exploit (e.g. remote timing attacks) from the others (e.g. locally exploitable attacks). When asked specifically about the library's threat models with respect to side-channel attacks, 20 libraries claimed remote attackers are in their threat model, 16 included local attackers, 1 included speculative execution attacks, 2 included physical attacks and 2 included fault attacks. Some libraries expressed that they consider some classes of attacks in their threat model if they are easy to mitigate, 2 would do so for local attacks and 1 for physical attacks. The general attitude towards side-channel attacks varied, 2 said that all side-channel attacks are outside their threat model and 10 said that their protections against side-channel attacks are best

effort. For example, one participant said: *"Best-effort constant-time implementations. CPU additions and multiplications are assumed to be constant-time (platforms such as Cortex M3 are not officially supported)."* (P2) Another one implied a progressive widening of their threat model regarding timing attack in their statement: *"Protections against remote attacks, and slow movement to address local side channels, though the surface is wide."* (P3)

In a follow up question, 23 libraries agreed that timing attacks were considered a relevant threat for the intended use of the library and its threat model, while this was not true for 2 libraries. We did not get this information for 2 libraries.

Reasons for considering timing attacks as relevant for their threat model were given as the ease of doing so (2), the threats of key-recovery in asymmetric cryptography (3), user demands (1), fear of reputation loss (1), use in a hostile environment (6), that attacks get smarter (1), the (rising) relevance of timing attacks (9), personal expectations (5), a connected environment (2), or the large scope of the library/of timing attacks (3).

Reasons for not considering timing attacks as part of their threat model were stated as this not being a goal of the library (2) or that they only consider more "practical" attacks.

2) Resistance against timing attacks (RQ1b): Here, we answer the research question whether libraries claim resistance against timing attacks (RQ1b). Many libraries do not have a systematic approach to address timing attacks; they only consider fixing "serious" vulnerabilities that could be exploited in practice. This might result in vulnerable code that can be exploited later with better techniques of recovering leaking information. We also encountered differing answers of different participants regarding suitedness of random delays as a mitigation. Out of the 27 total libraries, 13 claimed resistance against timing attacks. An additional 10 claimed partial resistance, 3 claimed no resistance, and for 1, we obtained no information.

We also asked how the development team decided to protect against timing attacks. For 4 libraries, participants reported that one person made this decision, for 12 it was a team decision, for 2 it was a corporate decision (where high-level management makes a decision or the team decided locally based on a corporate mission statement), for 14 libraries, participants reported that a priority trade-off caused their decision (e.g., lack of time to fully enact the decision) and 5 inherited the decision from previous projects or developers.

For 6 it was obvious that they needed to protect against timing attacks. For example, one participant stated: *"There was no decision, not even a discussion. It was totally obvious for everybody right from the start that protection against timing attacks is necessary."* (P4) Another one said: *"It's just how you write cryptographic code, every other way is the wrong approach (unless in very specific circumstances or if no constant-time algorithm is known)."* (P5) Another stated

> *"It became clear that these attacks transition from being an "academic interest" to a "real world problem" on a schedule of their own development. If something is noticed we now tend to favor elimination on first sight without waiting for news of a practical attack."* (P6)

Contrarily, another said:

> *"Basically a tradeoff of criticality of the algorithm vs practicality of countermeasures. Something very widely used (eg RSA, AES, ECDSA) is worth substantial efforts to protect. Something fairly niche (eg Camellia or SEED block ciphers) is more best-effort"* (P7)

This reasoning of waiting for attacks to justify expending the effort was also reported by another participant: *"For many cases there aren't enough real world attacks to justify spending time on preventing timing leaks."* (P8)

3) Timing attack protections (RQ2a): Here, we answer the research question how developers choose to protect against timing attacks (RQ2a). Developers address timing attacks in various ways, for example by implementing constant-time hacks (e.g. constant selecting), implementing constant-time algorithms of cryptographic primitives, using special hardware instructions (CMOV, AES-NI), scatter-gathering for data access, blinding secret inputs, and slicing. Many are interested and willing to invest effort into this - to various degrees, as P9 puts it: *"[T]hey're not that hard to mitigate, at least with the compilers I'm using right now"* (P9). Others are deterred by the lack of (easy-to-use) tooling.

We asked developers of the 23 libraries who considered timing attacks at least partially if and how their library protects against timing attacks.

For 2 libraries, participants reported that they use hardware features (instead of leak-prone algorithms) that protect from timing attacks such as AES-NI. For example, P7 said: *"AES uses either hardware support, Mike Hamburg's vector permute trick, or else a bytesliced version."* (P7)

For 21 libraries, participants said that they use constant-time code practices, which should in theory mean that code is constant-time by construction, but may be vulnerable to timing attacks after compilation. For example, P2 explained that: *"Conditional branches and lookups are avoided on secrets. Assembly code and common tricks are used to prevent compiler optimizations."* (P2)

For 9 libraries, participants explained that they choose known-to-be constant-time algorithms, but may suffer from miscompilation issues and end up non-constant-time. As an example, P7 said: *"If I know of a "natively" const time algorithm I use it (eg DJB's safegcd for gcd)."* (P7)

For 7 libraries, participants said they use "blinding", which means using randomization to "blind" inputs on which computation is performed, thereby destroying the usefulness of the leak. As P7 said: *"If blinding is possible [...] it is used, even if the algorithm is otherwise believed constant-time. "* (P7)

For 2 libraries, participants said that they protect through bitslicing, i.e., the implementation uses parallelization on parts of the secrets, hiding leaks. As one participant described: *"For instance, the constant-time portable AES implementations use bitslicing."* (P11)

For 2 libraries, participants reported protecting by "assembly", i.e., they have a specialized low-level implementation for protecting against compilers doing non-constant-time transformations. One participant noted the prohibitive cost of this practice, explaining:

> *"We do not write all constant-time code in assembly because of the cost of carrying assembly code. It is possible that the compiler may break the constant-time property. We spot-check that using Valgrind."* (P12)

For 1 library, timing leaks are made harder to detect by adding random delays.

Most developers focus on asymmetric crypto. Some do not consider old primitives, such as DES, which is still used in payment systems as Triple-DES. For 5 libraries, participants stated that they only protect a choice of modules: those libraries have multiple implementations, of which only some might be constant-time, maybe even insecure by default.

> *"Legacy algorithms like RC4 and DES are out of scope. If you use the <libraries'> "BIGNUM" APIs to build custom constructions, it's probably leaky, since bignum width management is complex."* (P13)

also mentions bignum libraries being specifically hard to secure. This claim is supported by academic literature as well: *"lazy resizing of Bignumbers in OpenSSL and LibreSSL yields a highly accurate and easily exploitable side channel"* [77].

For 1 library, protection against timing attacks was reported to be still in progress, e.g., they try to use constant-time coding practices throughout the library, but this is still in development due to large legacy code base.

> *"All decisions in a side project are limited by the available resources. There's a report about a new attack which proposes a new counter-measure: Does someone have the time to implement it? Yes - cool, let's do it. No - fine, let's put it on the ToDo list."* (P15)

and *"Very early on in its development these guarantees were much weaker, and in a few cases, approaches were used that turned out to be known to be imperfect."* (P16) were two answers from participants of libraries being in very different phases of solving this problem.

4) Testing of timing attack resistance (RQ2b, RQ2c): In Software Engineering, testing code for the properties it should achieve is commonplace and generally considered best practice [78]. We therefore were interested in the practice of testing and verification for constant-time also. Here, we answer the research questions whether, how, and how often libraries test for/verify resistance against timing attacks (RQ2b, RQ2c).

For 21 libraries, at least some type of testing was done, of which 14 were fully, and 7 were partially tested. 6 were not tested including the 2 libraries which claimed timing attacks are not relevant. 24 personally tested their libraries.

Of those, 12 stated they have tested manually, and 11 stated they tested automatically. Those two answers are not exclusive, since 7 libraries which test code automatically have also been tested manually. For manual testing, 6 libraries analyzed (parts of) their source code, 4 libraries analyzed (parts of) their binary, 5 did manual statistical runtime testing for leakage, and 1 ran the code and looked at execution paths, debugging as it ran. *"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"* (P17) conveys the experience quite graphically.

For those who did automated testing, 9 libraries used a Valgrind-based approach, 2 used ctgrind, 1 used Memsan, 1 used TriggerFlow, 1 used DATA, and 1 reported automated statistical testing without specifying further.

For the participants who did at least partial testing for resistance to timing attacks, we asked for testing frequency. For 1, the testing was only done once. For 11, participants reported manual or occasional testing. For 4, participants reported testing on release. For 6 libraries, participants reported that testing for resistance to timing attacks was part of their continuous integration. For 11 libraries, we did not obtain information on testing or testing frequency. These varying answers suggest that despite a common awareness of timing attacks, cryptographic developers never came to a consensus on the best way to address timing attacks in practice.

5) Tool awareness (RQ3a): In order to effectively test, developers should be able to leverage existing tooling created for the purpose of testing and/or verifying that source code, or compiled code, runs in constant time. Here, we answer the research question whether participants are aware of the existence of such tooling (RQ3a). We asked participants whether they are aware of tools that can test or verify resistance against timing attacks, also showing them a list of tools from Table I. We asked them whether they had heard about any of those tools with regards to verifying resistance against timing attacks. Table III shows the results with 33 being aware of at least one tool and 11 being unaware of any tool. ctgrind was most popular (27 heard of it; 17 had tried to use it), followed by ct-verif (17 heard of it; only 3 tried to use it) and MemSan (8 heard of it; 4 tried to use it). DATA had been used by 2, all others by no more than 1. Individual tool awareness and use numbers can be found in Appendix B.

For those tools they had heard about, we asked them where they had heard about them. Overall, participants were recommended a tool by a colleague 33 times, heard of a tool from its authors 20 times, read the paper of the tool 27 times, read about the tool in a different paper or blog post 42 times and heard of it some other way 24 times. 2 were involved in a development of a tool. A general tool they are already using can also be used for constant-time-analysis, which P18 learned through our survey:

> *"I already use MemSan primarily for memory fault detection. Was not aware of its use for side-channel detection but will try it in future since it is already integrated with my workflow to some extent."* (P18)

Again for the tools they were aware of, we asked which (if any) they had (tried to) use in the context of verifying or testing resistance to timing attacks. Table III displays the results, with 19 having tried to use at least one tool and 25 having never tried any of the tools.

6) Tool experience and use cases (RQ3b): Here, we answer the research question which experiences participants made with tools (RQ3b). As we were anecdotally aware that tools may be hard to obtain, unmaintained, and may be closer to research artifacts than ready-to-use tooling, we were interested in participants' experiences, finding that experience varied by tool, use cases and expectations. We therefore asked participants to describe the process of using the tools.

12 reported that they managed to get the respective tool to work at least once, but not necessarily repeatedly, while 3 reported that the tool they attempted to use failed to work even once, for various reasons, including excessive use of resources, such as effort, time, RAM, CPU cores, machines etc. One participant said of the DATA tool: *"it uses a ridiculous amount of resources"* (P17).

2 reported that they had integrated the tool into CI and were using it automatically. 12 reported that they used it manually, of which 6 said they use it during development, and 6 said they use it after development, on release. A participant said: *"Periodically, and manually, used when altering / writing code to check constant-time property."* (P12)

For those who had heard of specific tools, but had not attempted to use them, we were also interested in their reasoning. The reasons were varied, many including a lack of resources such as time (26) or RAM, CPU cores and machine (1). Participants also reported on bad availability (4), and maintenance (5), as well as insufficient language support (4), and other usability issues, such as problems with setting up the tool (3), or getting it to work properly post setup (1). The difficulty or impossibility of fulfilling the required code changes, such as markup for secret/public values, memory regions/aliasing, and additional header files was also a problem (reported by 1), as was the inability to ignore reported issues, once flagged by the tool (8).

Some reported not needing the respective tool (22), using other tools (18), gave reasoning that to our understanding was based on misconceptions of the respective tool (2), or reported having been unaware of the tool's capabilities in the context of resistance to timing attacks (1).

One participant also said that the tool was also used to verify a security disclosure. *"Tried to use to reproduce results, verify disclosures. Tried to use it to discover new defects in existing code."* (P14) — since the tool is later stated as in use by another member of the same project, this confirms that the tool not only verified the initial defect, but works as planned.

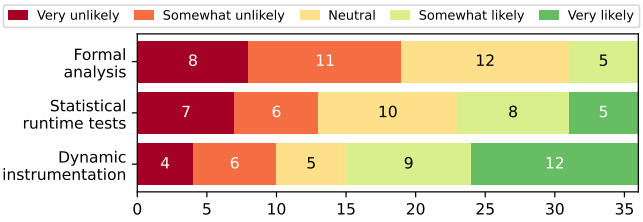

Fig. 3. Reported likeliness of tool use based on requirements and guarantees.

7) Potential Tool use (RQ4): In addition to understanding participants' current threat models and behaviors concerning constant-time code, we were also interested in what they thought about potential future use of testing/verification tools, and whether they would potentially be willing to fulfill certain requirements in exchange for guarantees (RQ4, see Figure 3). Generally, they were most willing to use dynamic instrumentation tools, and also spoke about them the most positively,

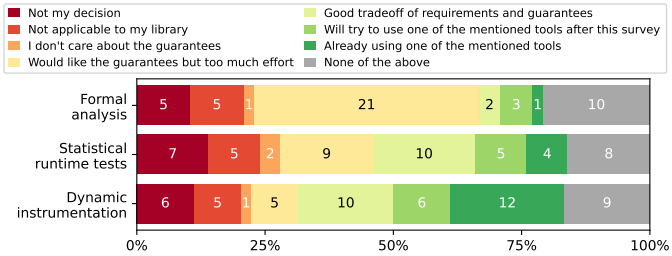

Fig. 4. Participant reasoning behind their likelihood of tool use.

whereas they mostly mentioned drawbacks when asked about formal analysis tools.

We presented the participants with the requirements and guarantees offered by three categories of tools: dynamic instrumentation based tools, statistical runtime tests and formal analysis tools.[5] We then asked them to rate their likeliness of using the presented group of tools on a 5-point Likert scale from "1=very unlikely" to "5=very likely". Figure 3 shows a strong preference for dynamic tools, while formal analysis tools are least likely to be used. We perform statistical tests on these ratings to establish that these differences are statistically significant. We find a significant difference in participants' self-reported likeliness to use tools in the different categories (Friedman Test-Statistic=18.477, p<0.0001). Post-hoc testing showed that participants are significantly more likely to use dynamic instrumentation based tools like ctgrind (mean=3.53, sd=1.38) than statistical tools (mean=2.94, sd=1.31; p=0.023, Durbin-post-hoc (DPH), Benjamini-Hochberg-corrected (BH)) and formal analysis tools (mean=2.38, sd=0.98; p<0.0001, DPH, BH-corrected). The difference between statistical and formal analysis tools was not significant (p=0.18, DPH, BH-corrected). Specifically, while 21 reported being somewhat likely or very likely to use a dynamic testing tool for resistance to timing attacks in the future, 13 reported the same for statistical runtime test tools, and only 5 said they were somewhat likely to use formal analysis tools.

We also asked participants to clarify their reasoning by choosing explanations (see Figure 4). Results show that participants would like the guarantees formal analysis tools provide, but perceive them as requiring too much effort (21) compared to the other tools (9 statistical runtime tools, 5 dynamic instrumentation tools). More participants think that the trade-off of effort and guarantees is acceptable for dynamic (10) and statistical tools (10) than formal analysis tools (2). More details on participants' reasoning follow.

a) Dynamic Instrumentation Tools: For dynamic instrumentation tools, some participants were happy with the limited guarantees, understanding the trade-off clearly.

> *"We currently use MemSan and Valgrind because they have very low maintenance since they pretty much come with the operating system, and we could get useful results from them with a few days' work. We are aware of their limitations (they miss non-constant-time parts, and*

[5]For the survey questions see the Appendix sections A-E,A-F and A-G.

of course they can only test code in the conditions where it is executed as part of the tests)." (P19)

The approach taken by tools like ctgrind is understandable to developers, so much so that some came up with it independently: *"We independently came up with this approach and were using it [before we] knew ctgrind existed."* (P9)

One participant specifically commented on the effort required to create and maintain annotations:

> *"A thing this survey might be underestimating is also the cost of code annotations: it's not just about having someone annotating the code properly (which already is quite a lot of effort) but there might be resistance for inclusion of such annotations in the code base as they add a maintenance burden for the project. Maintainers should fully understand the notation syntax and get proficient in it to spot instances where annotations need to be updated, moved, etc."* (P20)

b) Statistical Tools: For tools based on statistical tests of the runtime, 7 expressed that the guarantees provided by the tools are limited. One participant explained:

> *"I am dubious that it would provide much value over existing mechanisms. Also, CI currently runs on shared hosts which are timing noisy. From this noise I would expect [...] false positives [...]"* (P21)

Another participant also expressed concern over the guarantees and false positives / negatives: *"The requirements seem straightforward, but a statistical test seems likely to cause both false negatives and false positives."* (P13)

c) Formal Analysis Tools: Participants had strong feelings about the lack of usability of formal analysis tools:

> *"I'm very interested in these sorts of tools, but so far it seems formal analysis tools (at least where we've tried to apply it to correctness) are not really usable by mere mortals yet. I would be happy to be proven wrong, however!"* (P13)

The fact that compiler optimizations can introduce timing leaks that will not be detected by tools working at the source code level was highlighted by a few participants: *"Static analysis on the source code in most programming languages is NOT sound: it misses compiler optimizations that introduce secret-dependent flows."* (P19) and another one explaining *"I'm much more worried about compilers failing to preserve constant-time code, ..."* (P13)

While 4 mentioned their expectation of a higher effort to create the necessary markup for formal analysis-based tools, expectations of the scalability of these tools seem to be in line with other categories of tools.

Additionally, we found that participants were intimidated by the theory-heavy approach by formal analysis-based tools, thinking of formal verification in general. *"I have no experience with formal verification toolchains"* (P23).

More academically focused formal analysis tools also suffer from a maintenance problem if the developers have moved on to other research: *"Who knows if the toolchain is still maintained in a year?"* (P5)

In conclusion, dynamic tools are mostly criticized for requiring code annotation, while statistics ones are viewed critically because of their poor guarantees. However, participants were most critical towards formal analysis tools. Some doubt that such tools would be maintained, or question that fact that they would provide large support for different platforms. While these drawbacks are real, they do concern all tools, but participants point them out mainly for formal analysis tools. Some participants mention that such tools have steep learning curves, as they are not only unfriendly to use, but they also require specific knowledge. We notice that developers using ctgrind took their time to explain how it actually works (they we were never asked to), while participants remain vague about formal analysis tools. Only one participant actually uses such a tool, only few have tried, but not succeeded. However, many qualify such tools as uneasy to use, inefficient, lacking wide support, unable to verify external code, based only on the code, hard to be CI automated, adding very little confidence, and possibly unmaintained in the future.

8) Misconceptions: Despite surveying an expert population of cryptographic library developers, our study pointed out some misconceptions and differences of opinion about constant-timeness, timing attacks, and verification/testing tools. Those may deter from analysis tool use, and may contribute to more hidden timing vulnerabilites, ultimately making it harder to solve the timing attack problem in practice.

Some participants seemed to think constant-time is easily achieved. This logic implies if a project has a timing vulnerability, they have made a basic mistake.

> *"Writing constant-time code, contrary to writing [...] memory-safe code, is not hard, if you do it explicitly from start (caveat: when there's Gaussian rejection sampling in a lattice system, it _is_ hard[...]"* (P11)

This ties in with code annotation not being usable when secretness of variables changes, specifically as mentioned with rejection sampling. This misconception is based around most common use of annotations. If the annotations allow for declassification of variables, this problem can be resolved granularly. Not all tools allow this, though, so the misconception that this is true for all tools may have taken root.

One participant suggested that they do not need to test code if they write constant-time code correctly. *"In that sense, the guarantees offered by these tools are not worth putting effort into running them, at least in the case of <library>, where all code was designed to be constant-time"* (P11). This sentiment comes with several problems: on the one hand, humans make mistakes, so testing code is a best practice in software engineering for precisely this reason. Additionally, compiling code that does fulfill the constant-time property may create problems, as the compiler may change the original control-flow while adding some optimizations.

While talking about compilation units and control-flow, a partial misconception can be found in verification scope: *"a lot of code will exist outside of the boundaries of the library. A project using <library> would be more likely to be successful."* (P20) While the library may not know which inputs are secret , looking at an API should make it clear which inputs can be secret, and the constant-time criterion could be tested for all of them without knowing the actual usage patterns.

Furthermore, the different answers about random delays and statistical analysis tools show that there is no universal consensus among the participants. A participant said: *"Anything involving secret data, and in particular private-key data, has the timing dithered and with throttling of repeated attempts to make attacks of this kind difficult."* (P24) We are skeptical about this due to the results of Brumley and Tuveri [15]. If a side channel signal is measured as a timing difference between executions, adding a random noise distribution to these executions will reproduce a similar difference if enough samples of the executions are obtained. This can be done in parallel from different sources or over a long time, going around the throttling defense. A more practical quote is from P9:*"We once tried to test actual execution timings, but it wasn't reliable. We no longer do that. Now we use Valgrind."*

Lastly, even if cryptography is rather heavy in mathematics, some participants associate math/formal analysis as a barrier to using tools from that research area. *"[P]roving things like loop bounds is often arcane. Also, it's knowledge that would present a barrier to new engineers joining the team."* (P12) This is most likely a misconception, potentially caused by unclear writing in formal analysis tools' documentation, or scientific publications that do not separate tool use from general formal verification and theorem proving.

9) Developer Concerns and Wishlist (RQ5): In addition to misconceptions, participants also voiced understandable concerns about constant-time development, as well as wishes for verification tools that would allow them to use these tools more effectively (RQ5). Major concerns were voiced about the tools' resource usage being too high (see Section IV-B6).

In addition to these issues, P14 listed concerns as: *"the execution time of static and dynamic analyzers tailored for SCA, the need for human interaction, the rate of false positives, etc. are usually preventing a systematic adoption"*. The issue with flagging false positives and not linking false positives and negatives was addressed by another participant also: *"We noticed a couple false positive, where there *is* a path from the contents of the buffer to timings, but we decided that doesn't leak any meaningful secret."* (P9). They also mentioned security concerns for tools based only on the source code. These may miss vulnerabilities due to miscompilation, as explained by P13: *"Any "constant-time" code is an endless arms race against the compiler"*.

Interestingly, participants had many precise ideas for what could be done to improve the status quo of testing/verification tools. For example, for better usability, they ask for the ability to ignore some issues and/or some part of the code, as noted by P14: *"Also, expect a lot of "noise" from BIGNUM behavior that is not CT and requires a full redesign to be fixed."*

We saw many wishes for improvements concerning annotations, asking for external annotations. Participants also asked for easy maintenance of code annotations (see IV-B7a), and requested that tools work on complex code, as P14 explained: *"even for expert users the chances of exposing something nonconsttime to remote attackers are high, especially given the complex nature of <library> under the hood."* They also asked for test cases to be fast to set up, to avoid a *"non-trivial amount of effort to set up comprehensive tests."* (P14)

To address the issue of scale, they want to be able to use tools in CI. Otherwise, when the code changes, the guarantees are lost. This means that error code outputs, easy CI setup and runtime are important, as explained by P19: *"Static analysis tools tend to have a high engineering overhead: getting the tool to run, deploying it to CI systems, maintaining the installation over the years."* Similarly, participants demanded that tools not require rewrites of their code: P2 ruled out an *"awesome tool"*, because it *"cannot verify existing code."* Participants also required no restricted language or environment for their code instead of *"a pretty special-purpose language"* (P26). Similarly, they asked for no use of a specialized compiler; as P4 stated: *"Requiring a dedicated compiler sounds like a potential problem."* Generally, they asked for integration into type system and APIs they are already using: *"which values are public and which private, we have flags on APIs to allow the caller to specify this too"* (P28), so the project already has a form of security annotations for the users of their API, which a tool should be able to integrate for its analysis.

They also requested long-term available source code and longterm maintenance. As P25 stated, tools being unavailable or unmaintained makes it impossible to use them.

V. Discussion

Based on our findings, we make suggestions for four groups of actors who can take action to make cryptographic code resistant to timing attacks: tool developers, compiler writers, cryptographic library developers, and standardization bodies.

A. Tool developers

In spite of the fact that we selected a subset of well-known tools from the wide diversity of available tools, 25% of the developers who answered our survey did not know about any of them. Some developers learned about the tools from our survey. Only 38.6% actually using any of the tools shows that their adoption is limited. This can be partially explained by the relative youth of the tools, as most tools are less than 5 years old. However, we believe that many other factors come into play: tools may be research prototypes that are difficult to install, not available or not maintained; they may not be evaluated on popular cryptographic libraries, raising concerns about applicability and scalability; they may be computationally intensive, making their use in CI unlikely; they may not be published in cryptographic engineering venues. In addition to the specific recommendations from the previous section, we recommend the community of tool developers to:

1) make their tools publicly available, easy to install, and well-documented. Ideally, tools should be accompanied with tutorials targeted to cryptographic developers; making a tool easier to install by providing Linux distribution packages lowers the barrier to adoption.
2) publish detailed evaluations on modern open-source libraries, creating or using a common set of benchmarks; Supercop [79] is one such established benchmark;

3) focus on efficient analysis of constant-timeness, rather than computationally expensive analysis of quantitative properties, which seem to be of lesser interest. Ideally, tools should be fast enough to be used in CI settings;
4) make their tools work on code with inline assembly and generated binaries to be fully usable by all developers.
5) promote their work in venues attended by cryptographic engineers, including CHES, RWC, and HACS.

Ultimately, we recommend tool developers to follow Reid et al.'s recent advice to "meet developers where they are" [71].

B. Compiler writers

Developers are very concerned that compilers may turn constant-time code into non-constant-time code. To avoid this issue, developers often use (inlined) assembly for writing primitives. This approach guarantees that the compiler will not introduce constant-time violations but may negatively affect portability and makes analysis more complex. In order to make integration of constant-time analysis smoother in the developer workflow, we recommend compiler writers to:

1) improve mechanisms to carry additional data along the compilation pipeline that may be needed by constant-time verification tools. This would allow cryptographic library developers to tag secrets in source code and use constant-time analysis tools at intermediate or binary levels;
2) support secret types, as used by most constant-time analyses, throughout compilation, and modify compiler passes so that they do not introduce constant-time violations, and prove preservation of the constant-time property for their compilers. This would allow cryptographic library developers to focus on just their source code;
3) more generally, offer security developers more control over the compiler, so that code snippets that implement a countermeasure (e.g. replacing branching statements on booleans by conditional moves) are compiled securely.

C. Cryptographic library developers

Cryptographic library developers are aware of timing attacks and most consider them part of their threat model. In order to eliminate timing attacks, we recommend library developers:

1) make use of tools that check for information flow from secrets into branch conditions, memory addresses, or variable-time arithmetic. Ideally the use of such tools is integrated into regular continuous-integration testing; if this is too costly, a systematic application of such tools for every release of the library may be a suitable alternative;
2) eliminate all timing leaks even if it is not immediately obvious how to exploit them. Attacks only get better and many examples of devastating timing attacks in the past exploited *known* leakages with just slightly more sophisticated attacks techniques;
3) state clearly which API functions inputs are considered public or secret. With a suitable type system, such information becomes part of the input types, but as long as mainstream programming languages do not support such a distinction in the type system, this information needs to be consistently documented. Doing so makes it easier to *use tools* for automated analysis and harder for programmers to *misuse* library functions due to misunderstandings about which inputs are actually protected.

D. Standardization bodies

A recent paper [8] advocates for the importance of adopting tools in cryptographic competitions, standardization processes, and certifications. We recommend that submitters are strongly encouraged to use automated tools for analyzing constant-timeness, and that evaluators gradually increase their requirements as constant-time analysis technology matures. Standardization bodies should try to avoid the use of cryptographic algorithms leaking timing information. In the case of Dragonfly Password Authenticated Scheme used in WPA3 by the Wi-Fi Alliance, many timing attacks have been discovered [80], [81] as the algorithm leaks timing information. However, many deterministic algorithms with no leaks are known [82].

VI. Conclusion

We have collected data from 44 developers of 27 cryptographic libraries, and analyzed the data to gain a better understanding of the gap between the theory and practice of constant-time programming. One main finding of our survey is that developers are extremely aware of and generally concerned by timing attacks, but currently seldom use analysis tools to ensure that their code is constant-time. While constant-time testing may not be the most important thing on cryptographic developers' to-do list, it should become best practice. We think that this is only feasible by making tools more usable, supporting developers' current workflows, requiring little work overhead, and giving easy-to-understand outputs. Based on our survey, we have identified recommendations for tool developers, compiler writers, cryptographic library developers, and standardization bodies. We hope that these different communities will take up our recommendations and collectively contribute to the emergence of a new generation of open-source cryptographic libraries with strong mathematical guarantees. Although our recommendations are stated for timing attacks, we believe that many of our recommendations remain valid in the broader setting of high-assurance cryptography. In particular, all our findings are directly applicable to the many ongoing efforts to protect against micro-architectural side channels, as summarized in [60]. Another interesting topic would be a quantitative analysis of the usability of some of the better known tools collected in this study to gain insight into the exact magnitude of the mentioned usability problems.

Acknowledgements

This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE). J. Jancar was supported by Czech Science Foundation project GA20-03426S as well as by Red Hat Czech. Daniel De Almeida Braga is funded by the Direction Générale de l'Armement (Pôle de Recherche CYBER).

References

[1] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology – CRYPTO'96*, ser. LNCS, N. Koblitz, Ed., vol. 1109. Springer, 1996, pp. 104–113. [Online]. Available: http://www.cryptography.com/public/pdf/TimingAttacks.pdf

[2] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. ACM, 2003. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec03/tech/brumley/brumley.pdf

[3] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *the ACM Conference on Computer and Communications Security, CCS'12*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 305–316. [Online]. Available: https://doi.org/10.1145/2382196.2382230

[4] NIST, "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[5] M. Azouaoui, D. Bellizia, I. Buhan, N. Debande, S. Duval, C. Giraud, É. Jaulmes, F. Koeune, E. Oswald, F. Standaert, and C. Whitnall, "A systematic appraisal of side channel evaluation strategies," in *Security Standardisation Research - 6th International Conference, SSR 2020, London, UK, November 30 - December 1, 2020, Proceedings*, ser. Lecture Notes in Computer Science, T. van der Merwe, C. J. Mitchell, and M. Mehrnezhad, Eds., vol. 12529. Springer, 2020, pp. 46–66. [Online]. Available: https://doi.org/10.1007/978-3-030-64357-7_3

[6] A. Langley. (2010) ctgrind. [Online]. Available: https://github.com/agl/ctgrind

[7] J. Jancar. (2021) The state of tooling for verifying constant-timeness of cryptographic implementations. [Online]. Available: https://neuromancer.sk/article/26

[8] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1393, 2019. [Online]. Available: https://eprint.iacr.org/2019/1393

[9] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[10] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi, "Cryptanalysis of block ciphers implemented on computers with cache," in *Proceedings of the International Symposium on Information Theory and Its Applications, ISITA 2002*, 2002, pp. 803–806.

[11] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems – CHES 2003*, ser. LNCS, vol. 2779. Springer, 2003, pp. 62–76.

[12] D. J. Bernstein, "Cache-timing attacks on AES," 2005, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[13] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology – CT-RSA 2006*, ser. LNCS, vol. 3860. Springer, 2006, pp. 1–20.

[14] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password interception in a SSL/TLS channel," in *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*, ser. NCS, D. Boneh, Ed., vol. 2729. Springer, 2003, pp. 583–599. [Online]. Available: https://doi.org/10.1007/978-3-540-45146-4_34

[15] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security—ESORICS 2011*, ser. LNCS, V. Atluri and C. Diaz, Eds., vol. 6879. Springer, 2011, pp. 355–371, http://eprint.iacr.org/2011/232/.

[16] N. J. A. Fardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 526–540, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6547131.

[17] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on Amazon's *s2n* implementation of TLS," in *Advances in Cryptology – EUROCRYPT 2016*, ser. LNCS, M. Fischlin and J.-S. Coron, Eds., vol. 9665. Springer, 2016, pp. 622–643. [Online]. Available: https://eprint.iacr.org/2015/1129

[18] E. Brickell, "Technologies to improve platform security," Invited talk at CHES 2011, 2011. [Online]. Available: https://www.iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf

[19] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time RSA," *J. Cryptogr. Eng.*, vol. 7, no. 2, pp. 99–112, 2017. [Online]. Available: https://doi.org/10.1007/s13389-017-0152-y

[20] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015," in *Cryptology and Network Security*, ser. LNCS, S. Foresti and G. Persiano, Eds., vol. 10052. Springer, 2016, pp. 573–582. [Online]. Available: https://infoscience.epfl.ch/record/223794/files/32_1.pdf

[21] J. Jancar, V. Sedlacek, P. Svenda, and M. Sýs, "Minerva: The curse of ECDSA nonces; systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 4, pp. 281–308, 2020. [Online]. Available: https://doi.org/10.13154/tches.v2020.i4.281-308

[22] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, Gauss, and Reload – a cache attack on the BLISS lattice-based signature scheme," in *Cryptographic Hardware and Embedded Systems – CHES 2016*, ser. LNCS, B. Gierlichs and A. Poschmann, Eds., vol. 9813. Springer, 2016, pp. 323–345, https://eprint.iacr.org/2016/300/.

[23] P. Pessl, L. G. Bruinderink, and Y. Yarom, "To BLISS-B or not to be – attacking strongSwan's implementation of post-quantum signatures," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS'17*. ACM, 2017, https://eprint.iacr.org/2017/490/.

[24] G. Barthe, S. Belaïd, T. Espitau, P. Fouque, M. Rossi, and M. Tibouchi, "GALACTICS: gaussian sampling for lattice-based constant- time implementation of cryptographic signatures, revisited," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*. ACM, 2019, pp. 2147–2164. [Online]. Available: https://doi.org/10.1145/3319535.3363223

[25] T. B. Paiva and R. Terada, "A timing attack on the HQC encryption scheme," in *Selected Areas in Cryptography – SAC 2019*, ser. LNCS, K. G. Paterson and D. Stebila, Eds., vol. 11959. Springer, 2019, pp. 551–573. [Online]. Available: https://www.ime.usp.br/~tpaiva/papers/PaivaTerada_SAC2019_a_timing_attack_against_hqc.pdf

[26] G. Wafo-Tapa, S. Bettaieb, L. Bidoux, P. Gaborit, and E. Marcatel, "A practicable timing attack against HQC and its countermeasure," *Advances in Mathematics of Computation*, 2020. [Online]. Available: http://dx.doi.org/10.3934/amc.2020126

[27] Q. Guo, T. Johansson, and A. Nilsson, "A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on FrodoKEM," in *Advances in Cryptology – CRYPTO 2020*, ser. LNCS, D. Micciancio and T. Ristenpart, Eds., vol. 12171. Springer, 2020, pp. 359–386. [Online]. Available: https://eprint.iacr.org/2020/743

[28] D. Mayer and J. Sandin, "Time Trial: Racing Towards Practical Remote Timing Attacks," NCC Group, Tech. Rep., 2014, available at https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf.

[29] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, D. Atienza and G. D. Natale, Eds. IEEE, 2017, pp. 1697–1702. [Online]. Available: https://doi.org/10.23919/DATE.2017.7927267

[30] L. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1021–1038. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00074

[31] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation," in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, ser. LNCS, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., vol. 10492. Springer, 2017, pp. 260–277. [Online]. Available: https://doi.org/10.1007/978-3-319-66402-6_16

[32] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017,

pp. 235–252. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai

[33] T. Brennan, S. Saha, T. Bultan, and C. S. Pasareanu, "Symbolic path cost analysis for side-channel detection," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 27–37. [Online]. Available: https://doi.org/10.1145/3213846.3213867

[34] S. He, M. Emmi, and G. F. Ciocarlie, "ct-fuzz: Fuzzing for timing leaks," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 466–471. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00063

[35] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1767–1784. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/weiser

[36] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA - differential address trace analysis: Finding address-based side-channels in binaries," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 603–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[37] UCSD PLSysSec. haybale-pitchfork. [Online]. Available: https://github.com/PLSysSec/haybale-pitchfork

[38] MemorySanitizer. [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[39] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 161–173. [Online]. Available: https://doi.org/10.1145/3274694.3274741

[40] M. Neikes. Timecop. [Online]. Available: https://www.post-apocalyptic-crypto.org/timecop/

[41] P. Cuoq. tis-ct. [Online]. Available: http://web.archive.org/web/20200810074547/http://trust-in-soft.com/tis-ct/

[42] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," *Sci. Comput. Program.*, vol. 78, no. 7, pp. 796–812, 2013. [Online]. Available: https://doi.org/10.1016/j.scico.2011.10.008

[43] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds. ACM, 2017, pp. 362–375. [Online]. Available: https://doi.org/10.1145/3062341.3062378

[44] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed. USENIX Association, 2013, pp. 431–446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev

[45] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub, "Computer-aided verification for mechanism design," in *Web and Internet Economics - 12th International Conference, WINE 2016, Montreal, Canada, December 11-14, 2016, Proceedings*, ser. Lecture Notes in Computer Science, Y. Cai and A. Vetta, Eds., vol. 10123. Springer, 2016, pp. 279–293. [Online]. Available: https://doi.org/10.1007/978-3-662-54110-4_20

[46] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 77:1–77:29, 2019. [Online]. Available: https://doi.org/10.1145/3290390

[47] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: a DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 174–189. [Online]. Available: https://doi.org/10.1145/3314221.3314605

[48] B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 110–120. [Online]. Available: https://doi.org/10.1145/2892208.2892230

[49] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, ser. LNCS, P. Madhusudan and S. A. Seshia, Eds., vol. 7358. Springer, 2012, pp. 564–580. [Online]. Available: https://doi.org/10.1007/978-3-642-31424-7_40

[50] K. Athanasiou, B. Cook, M. Emmi, C. MacCárthaigh, D. Schwartz-Narbonne, and S. Tasiran, "Sidetrail: Verifying time-balancing of cryptosystems," in *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*, ser. LNCS, R. Piskac and P. Rümmer, Eds., vol. 11294. Springer, 2018, pp. 215–228. [Online]. Available: https://doi.org/10.1007/978-3-030-03592-1_12

[51] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 875–890. [Online]. Available: https://doi.org/10.1145/3133956.3134058

[52] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 1267–1279. [Online]. Available: https://doi.org/10.1145/2660267.2660283

[53] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 15–26. [Online]. Available: https://doi.org/10.1145/3213846.3213851

[54] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," *CoRR*, vol. abs/2104.10749, 2021. [Online]. Available: https://arxiv.org/abs/2104.10749

[55] UCSD PLSysSec. pitchfork-angr. [Online]. Available: https://github.com/PLSysSec/pitchfork-angr

[56] S. Chattopadhyay and A. Roychoudhury, "Symbolic verification of cache side-channel freedom," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2812–2823, 2018. [Online]. Available: https://doi.org/10.1109/TCAD.2018.2858402

[57] M. Balliu, M. Dam, and G. L. Guernic, "Encover: Symbolic exploration for information flow security," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, S. Chong, Ed. IEEE Computer Society, 2012, pp. 30–44. [Online]. Available: https://doi.org/10.1109/CSF.2012.24

[58] M. Nemec, D. Klinec, P. Svenda, P. Sekan, and V. Matyas, "Measuring popularity of cryptographic libraries in internet-wide scans," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, 2017.

[59] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[60] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for spectre defenses," *CoRR*, vol. abs/2105.05801, 2021. [Online]. Available: https://arxiv.org/abs/2105.05801

[61] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00011

[62] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for

the new spectre era," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 913–926. [Online]. Available: https://doi.org/10.1145/3385412.3385970

[63] 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. Z. Béguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 983–1002. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00114

[64] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1789–1806. [Online]. Available: https://doi.org/10.1145/3133956.3134043

[65] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 965–982. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00028

[66] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1807–1823. [Online]. Available: https://doi.org/10.1145/3133956.3134078

[67] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 154–171.

[68] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: Supporting developers in using cryptography," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.

[69] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, 2019.

[70] J. M. Haney, M. Theofanos, Y. Acar, and S. S. Prettyman, ""we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products," in *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, 2018, pp. 357–373.

[71] A. Reid, L. Church, S. Flur, S. de Haas, M. Johnson, and B. Laurie, "Towards making formal methods normal: meeting developers where they are," *arXiv preprint arXiv:2010.16345*, 2020.

[72] G. B. Willis, *Cognitive interviewing: A tool for improving questionnaire design*. sage publications, 2004.

[73] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[74] N. McDonald, S. Schoenebeck, and A. Forte, "Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–23, 2019.

[75] W. J. Conover, *Practical nonparametric statistics*. John Wiley & Sons, 1998, vol. 350.

[76] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal statistical society: series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[77] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big numbers - big troubles: Systematically analyzing nonce leakage in (ec)dsa implementations," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1767–1784. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/weiser

[78] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 85–103.

[79] "ebacs: Ecrypt benchmarking of cryptographic systems," accessed November 5, 2009. [Online]. Available: https://bench.cr.yp.to

[80] M. Vanhoef and E. Ronen, "Dragonblood: Analyzing the dragonfly handshake of WPA3 and eap-pwd," in *2020 IEEE Symposium on Security and Privacy, SP 2020*. IEEE, 2020, pp. 517–533. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00031

[81] D. D. A. Braga, P. Fouque, and M. Sabt, "Dragonblood is still leaking: Practical cache-based side-channel in the wild," in *ACSAC '20: Annual Computer Security Applications Conference*. ACM, 2020, pp. 291–303. [Online]. Available: https://doi.org/10.1145/3427228.3427295

[82] A. Shallue and C. E. van de Woestijne, "Construction of rational points on elliptic curves over finite fields," in *Algorithmic Number Theory, 7th International Symposium, ANTS-VII*, ser. LNCS, F. Hess, S. Pauli, and M. E. Pohst, Eds., vol. 4076. SV, 2006, pp. 510–524. [Online]. Available: https://doi.org/10.1007/11792086_36

APPENDIX A
SURVEY

A. Background

Q1.1: How many years have you been developing cryptographic code?

[Numeric field]

Q1.2: What background do you have in cryptography?

- ☐ Academic
- ☐ Took some classes
- ☐ On the job experience
- ☐ Teach it
- ☐ Hobby
- ☐ Industry
- ☐ Prefer not to say

Q1.3: Can you tell us a little bit more about your background as a developer who works on cryptographic libraries/primitives?

[Free text field]

B. Library / Primitive

Q2.1: What's your role in the development of *library*? (E.g., maintainer, project lead, core developer, commit rights, no rights, etc.)

[Free text field]

Q2.2: How are you involved in design decisions (e.g., concerning the API, coding guidelines and style, security-relevant properties) for *library*?

[Free text field]

Q2.3: What are the intended use cases of *library*? (E.g., embedded use, servers, etc.)

[Free text field]

Q2.4: What is the threat model for *library* with regards to side-channel attacks? (E.g., local/remote attackers, etc.)

[Free text field]

Q2.5: Do you consider timing attacks a relevant threat for the intended use of *library* and its threat model? Please give a brief explanation for why / why not. (If the execution time of a program depends on secret data, a timing attack recovers information about the secret by computing the inverse of this dependency. The two most notorious sources for such dependencies are secret dependent control flow and secret-dependent memory access. Timing attacks include cache attacks where

the attacker uses the cache to infer information about memory accesses of a target.)

[Free text field]

Q2.6: Does *library* claim resistance against timing attacks?

- ○ Yes
- ○ No
- ○ Partially
- ○ I don't know
- ○ Not yet but planning to

Q2.7: How did the development team decide to protect or not to protect against timing attacks? (We are interested in the decision process and not the protection mechanisms themselves (if any).)

[Free text field]

Q2.8: [only shown if **Q2.6** is "Yes" or "Partially"] How does *library* protect against timing attacks?

[Free text field]

Q2.9: Did you personally test for or verify the resistance of *library* against timing attacks?

- ○ Yes
- ○ No
- ○ Partially
- ○ Not yet but planning to
- ○ Not me but someone did
- ○ I don't know
- ○ Prefer not to say

Q2.10: [only shown if **Q2.9** is "Yes" or "Partially"] How did you test or verify the resistance against timing attacks? (E.g. using which tools, techniques, practices.)

[Free text field]

Q2.11: [only shown if **Q2.9** is "Yes" or "Partially"] How often do you test or verify the resistance of *library* against timing attacks?

- ☐ Only did it once
- ☐ Do it occasionally
- ☐ During releases
- ☐ During CI
- ☐ Don't know
- ☐ Prefer not to say

C. Tooling

Q3.1: Are you aware of tools that can test or verify resistance against timing attacks?

- ○ Yes
- ○ No

Q3.2: Please tell us which of these you've heard of with regards to verifying resistance against timing attacks.

[List of tools from Table I.]

Q3.3: How did you learn about them? (Check all that apply)

[Matrix question with subquestions being the tools the participant selected in **Q3.2** and the following answer options:]

- ☐ Recommended by colleague
- ☐ Heard from authors
- ☐ Read the paper
- ☐ Referenced in a blog/different paper
- ☐ Was involved in the development
- ☐ Other

Q3.4: Which of these (if any) have you tried to use in the context of resistance against timing attacks?

[Multiple choice question among the tools selected by the participant in **Q3.2**.]

Q3.5: Why have you not tried to use these?

[Multiple free text fields for all of the tools the participant did select in **Q3.2** but not in **Q3.4**.]

D. Tool use

[All of the questions in this group are matrix questions with subquestions for all of the tools the participant did select in **Q3.2** and **Q3.4**, i.e. those tools that the participant knows and tried to use.]

Q4.1: Please describe the process of using the tools.

[Free text field]

Q4.2: I was satisfied with the installation process. (Please rate your agreement with the above statement.)

- ○ I quit using the tool before I got to this point
- ○ I quit using the tool because this was a problem
- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree or disagree
- ○ Agree
- ○ Strongly agree

Q4.3: I was satisfied with the prerequisites that the tool needed to work with my code. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

Q4.4: In my understanding the tool is sound. (Please rate your agreement with the above statement. A sound tool only deems secure programs secure, thus has no false negatives.)

[Same answer options as **Q4.2**]

Q4.5: In my understanding the tool is complete. (Please rate your agreement with the above statement. A complete tool only deems insecure programs insecure, thus has no false positives.)

[Same answer options as **Q4.2**]

Q4.6: I understood the results the tool provided. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

Q4.7: I was satisfied with the documentation of the tool. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

Q4.8: I was satisfied with the overall usability of the tool. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

Q4.9: I was satisfied with the tool overall. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

E. Tool use: Dynamic instrumentation based

Q5.1: Use of dynamic instrumentation based tools like ctgrind, MemSan or Timecop requires:

- Creating test harnesses.
- Annotating secret inputs in the code.
- Compiling code with a specific compiler (in the MemSan case).

and in return detects non-constant time code that was executed (e.g. branches on secret values, or secret-dependent memory accesses). However, it does not detect non-constant time code that was not executed (in branches not executed due conditions on public inputs).

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q5.2: Can you clarify your reasoning for the answer?

- ☐ Not my decision
- ☐ Not applicable to my library
- ☐ Would like the guarantees but too much effort
- ☐ Good tradeoff of requirements and guarantees
- ☐ Already using one of the mentioned tools
- ☐ Will try to use one of the mentioned tools after this survey
- ☐ I don't care about the guarantees
- ☐ None of the above

Q5.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

F. Tool use: Statistical runtime tests

Q6.1: Use of runtime statistical test-based tools like dudect requires:

- Creating a test harness that creates a list of public inputs and a list of representatives of two classes of secret inputs for which runtime variation will be tested.

and in return provides statistical guarantees of constant-timeness obtained by running the target code many times and performing statistical analysis of the results.

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q6.2: Can you clarify your reasoning for the answer?

[Same answer options as **Q5.2**]

Q6.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

G. Tool use: Formal analysis

Q7.1: Use of formal analysis-based tools like ct-verif requires:

- Annotation of the secret and public inputs in the source code.
- Running the analysis via a formal verification toolchain (i.e. SMACK).
- Might not handle arbitrarily large programs or might require assistance in annotation of loop bounds.

and in return provides sound and complete guarantees (no false positives or negatives) of constant-timeness (e.g. no branches on secrets or secret-dependent memory accesses or secret inputs to certain instructions).

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q7.2: Can you clarify your reasoning for the answer?

[Same answer options as **Q5.2**]

Q7.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

H. Miscellaneous

Q8.1: Do you have any other thoughts on timing attacks that you want to share?

[Free text field]

Q8.2: Do you have any other thoughts on or experiences with those tools that you want to share?

[Free text field]

Q8.3: Do you have any feedback on this survey, research, or someone you think we should talk to about this research (ideally an email address we could reach)?

[Free text field]

Q8.4: Do you want to allow us to contact you for:

- ☐ sending you a report of our results from the survey
- ☐ asking possible follow-up questions

Q8.5: [Only shown if some of the options in **Q8.4** was selected] To allow us to contact you, please enter your preferred email address. (If at any time you want to revoke consent to contact you and ask us to delete your email address, please email [de-identified for submission])

[Free text field]

APPENDIX B
TOOL AWARENESS

Tool	Aware	%	Tried to use	%
ctgrind [6]	27	61.4%	17	38.6%
ct-verif [45]	17	38.6%	3	6.8%
MemSan [38]	8	18.2%	4	9.1%
dudect [29]	8	18.2%	1	2.3%
timecop [40]	8	18.2%	1	2.3%
ct-fuzz [34]	7	15.9%	1	2.3%
CacheD [32]	6	13.6%	1	2.3%
FaCT [47]	6	13.6%	0	0.0%
CacheAudit [44]	5	11.4%	0	0.0%
FlowTracker [48]	4	9.1%	1	2.3%
SideTrail [50]	3	6.8%	0	0.0%
tis-ct [41]	3	6.8%	0	0.0%
DATA [35], [36]	2	4.5%	2	4.5%
Blazer [43]	2	4.5%	0	0.0%
BPT17 [31]	2	4.5%	0	0.0%
CT-WASM [46]	2	4.5%	0	0.0%
MicroWalk [39]	2	4.5%	0	0.0%
SC-Eliminator [53]	2	4.5%	0	0.0%
Binsec/Rel [30]	1	2.3%	0	0.0%
COCO-CHANNEL [33]	1	2.3%	0	0.0%
haybale-pitchfork [37]	1	2.3%	0	0.0%
KMO12 [49]	1	2.3%	0	0.0%
Themis [51]	1	2.3%	0	0.0%
VirtualCert [52]	1	2.3%	0	0.0%
ABPV13 [42]	0	0.0%	0	0.0%
None	11	25.0%	25	56.8%

TABLE III
TOOL AWARENESS AND USE