"These results must be false": A usability evaluation of constant-time analysis tools

Marcel Fourné1,4, Daniel De Almeida Braga2, Jan Jancar3, Mohamed Sabt2, Peter Schwabe4,5, Gilles Barthe4,6, Pierre-Alain Fouque2, and Yasemin Acar1,7

1Paderborn University, Paderborn, Germany
2Rennes University, CNRS, IRISA, Rennes, France
3Masaryk University, Brno, Czechia
4MPI-SP, Bochum, Germany
5Radboud University, Nijmegen, The Netherlands
6IMDEA Software Institute, Madrid, Spain
7George Washington University, Washington D.C., United States of America

Abstract

Cryptography secures our online interactions, transactions, and trust. To achieve this goal, not only do the cryptographic primitives and protocols need to be secure in theory, they also need to be securely implemented by cryptographic library developers in practice.

However, implementing cryptographic algorithms securely is challenging, even for skilled professionals, which can lead to vulnerable implementations, especially to side-channel attacks. For timing attacks, a severe class of side-channel attacks, there exist a multitude of tools that are supposed to help cryptographic library developers assess whether their code is vulnerable to timing attacks. Previous work has established that despite an interest in writing constant-time code, cryptographic library developers do not routinely use these tools due to their general lack of usability. However, the precise factors affecting the usability of these tools remain unexplored. While many of the tools are developed in an academic context, we believe that it is worth exploring the factors that contribute to or hinder their effective use by cryptographic library developers [61].

To assess what contributes to and detracts from usability of tools that verify constant-timeliness (CT), we conducted a two-part usability study with 24 (post) graduate student participants on 6 tools across diverse tasks that approximate real-world use cases for cryptographic library developers.

We find that all studied tools are affected by similar usability issues to varying degrees, with no tool excelling in usability, and usability issues preventing their effective use.

Based on our results, we recommend that effective tools for verifying CT need usable documentation, simple installation, easy to adapt examples, clear output corresponding to CT violations, and minimal noninvasive code markup. We contribute first steps to achieving these with limited academic resources, with our documentation, examples, and installation scripts 1.

1Installation scripts, tasks, documentation and codebook are provided as an artifact, see Footnote 3.

1 Introduction

Timing attacks [68] are side-channel attacks that measure program execution time to infer information about confidential data. They are practical and can be used by (remote) attackers to achieve full recovery of secrets including cryptographic keys [28]. This makes protection against timing attacks an important goal for developers of cryptographic libraries.

In his seminal work, Kocher [68] observes that making control flow and memory access independent of secret data can help protect programs against timing attacks. Over the years, this guideline has become known as the constant-time discipline, and has become a gold standard for cryptographic libraries. Unfortunately, constant-time programming can be error-prone, especially when programming under stringent efficiency constraints, as is the case for cryptographic libraries. In 2010, Langley developed ctgrind [75], a minimal patch to Valgrind for checking that crypto software is constant-time. Subsequently, the security community has developed a broad variety of tools for protecting against timing attacks. Two recent works [61] and [49] provide an overview of these tools, from complementary perspectives. Jancar et al. [61] conduct a survey about the use of constant-time analysis tools with 44 developers of 27 widely deployed open-source cryptographic libraries. Their survey shows that these developers do not leverage constant-time tools despite an interest in writing constant-time code. As reasons, they identify that tools are not ready-to use and their use therefore requires significant time and expertise. Geimer et al. [49] presents a systematic evaluation of five selected tools, and identifies several technical roadblocks for the usability of tools. In addition, both works provide a systematic classification of around 40 tools for checking constant-time, and provide recommendations for tool developers and users. Although both [61] and [49] provide valuable insights on these tools, an empirical study to corroborate and deepen their findings has been lacking.

Therefore, in this work, we aim to understand which factors support and hinder effective use of CT tools through an empirical usability investigation that analyzes participant
strategies while working with CT tools. Our investigation provides a complementary view on the issues discussed in [61]—which predates this work—and [49]—which was published after we completed the developer study. Our developer study is designed to provide deeper insight into usability requirements and how they influence their interaction with CT tools, to determine the features that tools should provide to achieve their full potential. Due to the broad range of tools, we designed a usability study with six CT tools. The participants of the study are 24 advanced CS students who had knowledge in cryptography (including about CT programming) and C programming. Our study comprises two phases: in the first phase, participants work through tasks escalating in difficulty while familiarizing themselves with a tool; in the second phase, they analyze real-world cryptographic libraries for CT-ness.

We identify usability issues that we group into seven categories that revolve around three high-level aspects: (1) required efforts to setup and start using the tool, (2) barriers and work overhead hindering the use of CT tools, and (3) functionality the developer wants in analysis to identify and fix problems. We aim to answer the following research questions:

RQ1: What are the pain points when trying to use CT tools?
A: Installation, setup for analytic use, and (long term) operationalization in a larger library context are challenges for effective CT tool usage.

RQ2: How helpful are the tools at discovering and fixing problems? Which tool properties help or hinder effective use?
A: Tools can help cut down the amount of work needed to analyze larger code bases rigorously, but if a tool is too much work to install and get to work, cryptographers might just “eyeball” the analysis without the tool. Meaningful documentation to get a tool working on simple examples effectively helps to overcome this.

RQ3: How can we support potential users in using the tools?
A: Easy setup, a set of simple examples to appropriate for the markup (which should be minimal and noninvasive to the source code), a tutorial on how to use the tool and get clear information from the output, and good general documentation were all found to be helpful.

Based on our findings, we suggest how the usability of CT tools can be improved to make CT analysis more accessible to developers. In summary, our contributions in this paper are:

- We concretize the problems mentioned by experts in the Jancar et al. survey [61] through a developer study with 24 newly trained potential crypto developers and publish the full procedure material (see Footnote 3) for replication.
- We offer a systematization of crypto developer workflow in using CT analysis tools, common to all 49 tools we found (see Table 3).
- We document pain points and their impact on crypto developer usage of CT analysis, giving an explanation on why the findings of Jancar et al. [61] are still prevalent.
- We propose what to consider during development of CT analysis tools by contrasting prior attempts.

**Supplementary material and disclosure.** We have communicated our results to the authors of the tools included in our study and made the artifacts available to them. We have received four responses; all four expressed interest, one said they plan to link to our study materials in their project. The supplementary material, including tutorials, installation guides, and codebooks is publicly available on a dedicated web page\(^1\) and as an artifact\(^2\).

2 Background & Related Work

We give an overview over the background and related work to this research by first discussing impacts of timing attacks on security, then describing CT development and CT analysis as defenses. For context, we also discuss a new generation of timing attacks that exploit microarchitectural features of CPUs, and the related efforts to protect against these attacks. Finally, we explain how a lack of consideration of human factors in cryptographic development can hinder widespread effective use of cryptography.

1. Timing attacks. Since Kocher’s introduction of side-channel vulnerabilities in 1996 [68], these threats have persisted despite significant efforts to address them. Considering the vast range of side-channel attacks, we will highlight a few pivotal moments with a focus on timing attacks. Kocher’s seminal work highlighted vulnerabilities in asymmetric cryptographic algorithms like RSA and DSS through “Timing Attacks”, emphasizing the potential for exploitation based on secret-dependent operation times. In 2002, Tsuono et al. [104, 105] expanded timing attacks to symmetric cryptography, noting vulnerabilities in MISTY1, DES, and suggesting AES being vulnerable to cache-timing attacks. Independent work by Bernstein [13] and Osvik et al. [87] confirmed these AES vulnerabilities. In 2003, Brunley and Boneh [28] revealed that these attacks could be conducted remotely via network timings. Subsequent vulnerabilities were discovered in the SSL/TLS libraries [3, 27, 29, 44] and on hardware-assisted defenses, such as Yarom et al.’s “CacheBleed” [125]. Kaufman et al. [66] also warned of persistent vulnerabilities post-compilation.

Despite these vulnerabilities and an emphasis on fixing them, side channels remain common in numerous platforms [19–21,46–48,79,108,109]. Some Common Criteria certified devices, despite their countermeasures, were found vulnerable [62]. Moreover, even recent post-quantum cryptographic efforts are affected [26,54,88,89,103,113].

2. Constant-time Analysis. In this paper, we focus on investigating usability aspects of tools that evaluate timing leakages.
of (cryptographic) software. However, it is worth pointing out that the tools we consider also differ on a technical level in at least four different ways:

First, depending on the approach taken by different tools, they give very different soundness guarantees. Static formal analysis can achieve full soundness with regards to some leakage model. Slightly weaker guarantees are offered by tools performing symbolic execution; these tools achieve soundness only up to certain upper bounds on loop length. Tools based on dynamic analysis typically work with symbolic secret data but concrete public data; they achieve soundness up to code coverage for the concrete public values of the test cases. Statistical analysis performs measurements on (large sets of) concrete public and secret data. The advantage is that this approach does not require any leakage model, but on the downside, it also does not provide any soundness guarantees.

Second, the tools work on different levels of compilation. We distinguish tools working on source level, on some intermediate level, or on binary level. An example for a source-level tool would be the information-flow type system implemented by the secret_integers crate⁴ in Rust. All tools we study (we will give detailed introductions later in Section 3.2) in this paper work on either intermediate-representation (IR) of the LLVM toolchain [90] or on binary level. Tools working on IR level are inherently limited in the sense that they are unable to find any leakages introduced by the compiler when translating from IR to binary [66, 98].

Third, the tools working on binary level differ in what architectures and extensions they support. In order to be used on production code, they need support not just for the core instruction sets of widely used architectures, but also for vector instructions and dedicated crypto extensions.

Finally—and here is where technical features overlap with usability—the tools differ in terms of performance. For example, for the analysis of Langley’s “donna64” implementation [74] of Curve25519 [14], the running time of just two of the tools we considered ranges between 0.38 and 225 seconds. This wide range may impact Continuous Integration (CI)/Continuous Deployment (CD) and developer workflows.

Table 3 presents the tools we found and categorized according to prior literature [60], appending a few tools previously not included; similar tables are found in [49, 61]. For each tool, we describe the target of analysis, the techniques used and whether the tools claim to provide some form of formal guarantees. We opted to err on the generous side of claimed soundness guarantees of each tool. For some tools the claims do not easily map to the soundness categories we discussed before, so we keep the unqualified “Other” category from the literature. As usual with this kind of classification, the categories are not exclusive, each tool may combine approaches in its design—we opted to continue with best-effort categorization like the established literature.

4See https://docs.rs/secret_integers/.

3. Microarchitectural side-channel attacks and defenses

While constant-time programming is still an important and increasingly standard baseline defense against software-visible side channels, research on more advanced microarchitectural attacks in the past few years has shown that this programming discipline is not a sufficient measure. This line of research started with the 2018 Spectre [67] and Meltdown [78] attacks, and has since identified multiple pathways for attacks that often—but not always—exploit speculative execution in modern CPUs. See, e.g., [70, 81, 86, 117].

The notion of constant-time can be extended to protections against more advanced microarchitectural attacks [67], leading to notions of speculative constant-time [31] or more generally of security with respect to a hardware/software leakage contract [58, 82, 83]. Many of the techniques used for analyzing constant-timeness can be extended to reason about speculative constant-time and related notions. In fact, there is already more than two dozen tools that analyze whether a program satisfies (some variant of) speculative constant-time. For an overview of these tools see [30, Fig. 2]; they generally suffer from similar usability issues as tools for constant-time.

Recent work [77, 110, 111] shows that aggressive optimizations used by modern CPUs to improve performance can lead to a new class of timing attacks. Many of the leakages are data-dependent and depend on prior execution history, making their detection extremely challenging. As a consequence, there is a strong incentive to develop analysis tools for checking the counterpart to constant-timeness; see [12, 45] for two very recent examples.

In both cases, we believe that the insights gained from [49, 61] and our work will provide valuable input for improving the usability of future tools in this space.

4. Human Factors in Cryptographic Development

There is a large body of work on human factors in cryptographic development. Acar et al. establishes in a 2017 study that poor usability of cryptographic libraries contributes to misuse and insecure code [1]. Haney et al. investigate the mindset of cryptography developers [55], and observe that some developers do not adhere to mainstream software engineering practices.

Krueger et al. developed a wizard for secure code snippets for specific cryptographic applications, evaluating its effectiveness and usability in a programming study [71–73].

In the specific context of constant-time tools, a study by Cazulig et al. [32] was carried out with over 100 students to understand the benefits of the FaCT tool introduced in the paper. The tool support by FaCT is found helpful for generating new code that is CT. In extension of this work, we include a diverse set of CT tools, documentations, tutorials, as well as open source libraries in our study.

Unfortunately, while previous research suggests that lack of usability prevents effective use of security tools [36, 40, 50, 94, 112], and specifically for CT [61], the question of how to improve the usability of these tools has been understudied [2].
3 Usability criteria and tool selection

In this section, we give a general description of our usability criteria, and explain how they impact users. In addition, we briefly introduce the six included CT tools in our study, organizing our presentation to inspect the previously defined criteria for each tool.

3.1 Usability criteria

The main purpose of our evaluation is to assess the usability of current CT tools, and identify features that impact effective use. To explain on Jancar et al. [61], we define criteria revolving around three features: (1) the effort required to setup and familiarize, (2) the work overhead for secret designation and target building, and (3) the quality of output to identify and fix problems.

We define our criteria following how users would perform the tasks related to CT analysis [102]: how users might interact with the tool, what information is given to the user, and how analysis outcome is presented to the user. The categorization of CT testing workflow steps was created from our expert team’s experience in building CT tools and using them on real-world projects, combined with insights gained from piloting the study. We developed the categorization after all of the study results were gathered.

Installation. Every tool needs to be installed before use. There are two broad ways of installing CT tools. Some come pre-built and bundled for a package manager or in a container. Others involve manual installation by either building from the source, or by grabbing the available binary from a release page. For the latter method, the developer will be in charge of managing the necessary dependencies manually. CT analysis tools mostly come as proof-of-concept artifacts. According to [57], only 3% of artifacts are distributed in containers, while 23% are pre-built and 70% must be compiled from source code. Therefore, we expect that the installation step of CT tools may be very challenging for numerous tools, especially because of unmaintained dependencies, also confirmed by Jancar et al. [61], who point out that libraries maintainers do not consider use of hard-to-install CT tools.

Familiarization. Documentation is intended to provide a high-level overview of the tool and offers technical details for expert users. Help materials also include tutorials and examples. In this criterion, we focus on how quickly new users become comfortable running a tool on simple programs.

Building and Secret Designation. CT tools provide a means to tag secret data. This is typically achieved via either code annotation or the creation of an external function wrapper. Many CT tools operate on instrumented binaries or some abstract intermediate representation that is designed for program analysis. Very often, this implies a custom building and linking process. Usability is negatively impacted whenever manual work is needed during this process. In other words, we look at how much tools modify a project to be analyzed: both in terms of code (for secret designation) and build workflow integration (for target generation). Little work overhead is commonly appreciated [64].

Analysis Runtime. Once the target is built, users can actually run the tool for CT analysis. Here, we look at two sub-criteria, the tool’s interface and its runtime. For a command-line interface tool, users may struggle with passing the right options. Importantly, tools are expected to yield results in an acceptable time frame. The longer the runtime of the analysis, the more difficult it is to integrate the analysis into the project workflow [64]. This problem hinders a feedback loop using CT analysis at coding time. This can be important both in CI workflows, which may have an upper time limit, and developer workflows, where each developer may only want to spend a small amount of time waiting for analysis results.

CT Problem Fixing. When the analysis is finished, CT tools display some output to direct the developer’s attention to detected issues. The purpose is to provide the developer with enough information to judge whether or not they care about the issue, and if yes, why the tool reports it. For example, it is not helpful if tools just display that there is an issue without any detail about the origin of the leakage. In addition, it is more productive for developers to be able to navigate and manage the list of reported issues. Otherwise, developers must linearly search through the (potentially large) list of results, making selective fixing more difficult.

Specialized Output Generation. To improve the experience of fixing problems, users might require customizing the generated analysis output. We introduce two features that we identify for CT tools. First, tools should also offer different verbosity in report details to avoid excess of information [50]. For example, a summary mode is beneficial in order to quickly skim the reported vulnerabilities to decide which one to inspect. Second, within the context of a CI pipeline, a delta report can be handy in assisting developers to determine whether a specific leakage has been correctly patched, and that the fix has not induced other leakage.

Reliability / False Positives. Ultimately, users need to trust the tool and its analysis. Therefore, any indication of potential false positives or missed issues could undermine user confidence, leading to tool abandonment. Solutions do not necessarily involve sound or complete tools, but also support for filtering user-supplied false positive patterns. This may help the user but can also lead to user filtering actually missing timing leaks, either mistakenly or lazily.

3.2 Tools

We selected six tools for use in our study: MemSan, timecop, dudect, ctverif, BINSEC/REL, and haybale-pitchfork. These
tools were primarily chosen to include a representative from each analysis type. The selection of the tools was made towards the end of 2022, therefore more recent tools were not considered. We prioritized tools well-recognized in the community, ideally those used by developers, gauging their reputation through a recent survey [61]. Out of the tools, 4 (ctverif, MemSan, dudect and timecop) are 4 out of top 5 most known tools in [61], with the top one being ctgrind, which we replaced with the functionally equivalent and still maintained timecop, haybale-pitchfork and BINSEC/REL were selected as representatives of other tool approaches. The number of tools was also constrained by participant numbers to ensure even distribution. At the end of the subsection we compare our choice of tools with the five tools chosen in [49].

MemSan [99]. MemSan is designed to leverage the Clang built-in memory sanitizer to dynamically analyze binaries for constant-time violations, thereby requiring Clang for installation (which is available in most Linux package managers). Clang sanitizers are well documented, but there is little documentation on how to use MemSan for CT analysis. Concerning secrets, users can declare private variables and/or memory regions containing secrets, and declassify variables within certain code sections if required. To run the tool, developers must compile the program with Clang, using the appropriate option to enable the memory sanitizer. All parts with no enabled sanitization are ignored—it is easy to get this wrong. Then, the analysis is performed by running the resulted binary. Note that only the executed code is analyzed, leading to different conclusions when running the same binary with different inputs. Indeed, code coverage is essential for MemSan. Upon the binary execution, errors will be displayed on branching or memory access indexing an annotated variable. The output details the path between the annotated variable and the cause of leakage. The output messages will be more related to the source code if the target is compiled in debug mode.

timecop [84]. Similar to MemSan, timecop relies on the Valgrind memcheck module [39] to dynamically analyze binaries for CT violations. Therefore, for installation, it solely requires Valgrind (which is available in most Linux package managers) and an additional C header file that must be downloaded from the project page. The timecop page also contains several tutorials and examples to smooth its first uses by beginners. To analyze code, users need to annotate private variables in the source code and may declassify variables within certain code sections if needed. There is no need for changes in the compilation chain. Concerning the analysis, users can simply run Valgrind on the binary as if they were searching for memory leaks. Valgrind will raise warnings for CT violations just like it would for the use of uninitialized memory in a branching or memory access. The output details the path between the annotated variable and the cause of leakage. timecop relies on the debug information to display the lines of code in its warnings. With its use in SUPERCOP [15], it is widely used.

dudect [91]. Installation for dudect is virtually non-existent as the tool is provided as a simple archive containing the C header file implementing it. The dudect documentation is rather limited. The tool operates via a black-box evaluation of a function, obviating the need for code annotation. The user, however, is required to implement an external wrapper in charge of setting the analysis parameters and options, as well as two functions to initialize the secret input classes and call the code to assess, respectively. Then, the target program must be compiled (with no custom build) and executed for analysis. The dudect approach is statistical, and it thus outputs values of statistics after code analysis. The output does not underlie any source leakage, but only some probabilistic conclusion about the target CTness.

tcverif [4]. The installation of tcverif presents a significant challenge, requiring undocumented versions of specific dependencies and manual patches across different projects, such as SMACK and Bam-Bam-Boogieman. Aside from the paper, there is no or little documentation available. As for secret designation, users must declare private and public variables and/or memory regions (arrays) containing secret or public inputs. In addition, they could declassify outputs, and assert the non-overlapping nature of these regions. The tool operation is straightforward, requiring only the source code file as input, in addition to the entry point to analyze. Thus, tcverif does not need any custom build. However, tcverif can process a C translation unit only when all the called functions inside are defined by other input files, otherwise it produces an unknown error. After a run, tcverif only highlights the leakage location in the source code, without a dependency chain of variables or memory locations that lead to each leaked secret. Surprisingly, tcverif may raise some warnings even after a successful run without CT violations. It is worth mentioning that tcverif, instead of making some approximate analysis, informs developers when it cannot conclude about some leakage, displaying inconclusive output.

BINSEC/REL [38]. BINSEC/REL comes as source code, an extension to the Binsec tool. Some dependencies, such as an SMT solver and the OCaml package manager, shall be installed manually, before compiling the project source available on GitHub. BINSEC/REL offers a comprehensive list of supported command-line options and numerous examples to start with. On the analyzed project, users shall employ markup declarations to annotate the source code, thereby designating public and private data. The analysis of BINSEC/REL operates over binaries. The version utilized in this study only supports ARM 32 and x86_32 architectures, necessitating the target to be compiled accordingly. This might require to add additional compiler flags, since in numerous compilers, the default mode supports 64-bit. Upon completion of the analysis, a report is produced including the number of CT violations and an assembler dump correlating with the violation location. The assembler dump does not point to the leaked secret, but only
to the instruction causing the leakage. Note that during our study, Binsec/REL received a major update that integrated the CT checking functionality into the main tool Binsec.

haybale-pitchfork [106]. Written in Rust, haybale-pitchfork can be installed from source using cargo, although its dependencies must be manually installed beforehand as documented on the project page, which includes multiple examples and different documentation materials. haybale-pitchfork runs its analysis over the LLVM intermediate representation. Thus, users need to modify the compilation chain to produce the corresponding LLVM bitcode of the target. Any symbol in the generated bitcode must be correctly resolved, or haybale-pitchfork stops the analysis, while printing a message raising “other errors”. Instead of relying on annotations to mark secrets, users are instructed to implement an external wrapper in Rust, in order to define an abstract signature of the target function. Here, each function parameter can be declared as public or secret using the appropriate Rust type. This wrapper also contains other configurations, such as the bitcode path to inspect. Users carry out the analysis by compiling and executing the Rust wrapper. haybale-pitchfork provides conclusive results, displaying the leakage origin whenever a CT issue is found, together with a tree path to the leaked secret.

Comparison with the tools of Geimer et al. [49]. Geimer et al. [49] explores five tools in depth: Abacus [9], Binsec/REL, ctgrind, dudect, and MicroWalk-CI [122]. Two of these tools (Binsec/REL and dudect) are also included in our study. As explained above, we selected timecop and MemSan over ctgrind, because the ctgrind patches are outdated and do not work with recent versions of Valgrind and the Linux kernel anymore. In contrast, timecop and MemSan can be seen as more usable versions of ctgrind. We did not select Microwalk-CI [122], because it was released after we had initiated our study. We also did not select Abacus, because its focus is quantitative information flow rather than constant-timeness. We included cverif for its strong correctness and coverage guarantees. We also included haybale-pitchfork, as an instance of a tool that covers both constant-time and speculative constant-time—however, to our knowledge, the tool was eventually not extended to speculative constant-time.

4 Methodology

In this section, we provide details on the procedure and structure of the study we conducted with (initially) 31 participants. We describe the experimental setup including choice of libraries, surveys, and experimental infrastructure. We also describe our coding process of qualitative data, including participant behavior and free-text responses, as well as the approach for statistical analysis of quantitative data, such as success measures and quantitative survey items. Finally, we explain our data collection and ethical considerations, and discuss the limitations of this work.

1. Recruitment and Participants. Due to the challenges of recruiting professional developers with security expertise, we targeted CS students for participation. We engaged Master’s and early PhD students from 5 universities across 4 countries, as recommended by [101]. From the 74 students we approached with an eligibility survey, 31 began the study. A tools assignment error led us to exclude one participant.

Eligibility Criteria. We only consider participants having the minimum knowledge necessary to run CT tools. We asked them to self-report their knowledge of the C programming language and the CT paradigm. Naturally, considering our usability criteria, we also verify that they had never used any of the tools that are part of this study. Finally, to distinguish our work from [61], participants needed to lack experience in working on production-quality cryptographic code.

Compensation. Participants were compensated with 200 euros on completion of the study in exchange for 16 hours of participation; this compares to the hourly payment for student research assistants in participating countries.

2. Study Procedure. After a short eligibility and demographics survey which preceded the study, we asked participants to assess code for vulnerability to timing attacks using a CT verification tool across ten tasks that escalated in complex-
During the study, participants were assigned ten successive repair tasks—tasks building upon tool-support specifics tested in prior tasks, which we will explain in detail—in which they were instructed to use a pre-installed CT analysis tool (tool 1) to identify whether a given code snippet is CT regarding a well-defined secret. If the code was not CT, participants were asked to fix it. The repair tasks represent textbook examples of secret-dependent branching and memory access, and their CT variant. After working on the first task, participants were given a tutorial that we had written for the tool. After the second task, we gave them the solutions to previous tasks, to be used as examples. Tasks 3 to 8 added various elements to increase difficulty, such as calls to libc functions (memcmp), reading randomness from the operating system and particular source code designed to trigger optimization during compilation. The goal of these repair tasks is to assess the participants’ ability to use the tool to evaluate and fix a rather simple code snippet.

After completing work on the ten repair tasks (or exhausting the allotted time of 8 hours), and so becoming familiar with the tool, we asked them to audit well-known cryptographic libraries using the same tool (tool 1). In this audit task, participants were asked to compile the library (library 1) in such a way that enables them to use the tool, and audit a much larger code basis. They were pointed at potentially interesting parts of the library, but not at specific functions. After a first library audit with a tool they had used for the entire study up to that point, we provided them with a new tool (tool 2), a tutorial, and a new library (library 2) to start a second audit task. These audit tasks aim at assessing the tools’ usability in a setting more closely resembling a real-world use case.

For both parts of the study, we consider that a participant successfully completed a task if they underline the CT violation using the respective CT tool, and recognize it as such to fix it. The task structure was monolithic, simply stating that the task was to find CT violations with the given tool. Participants had to find out the necessary steps themselves.

After each of the repair tasks, participants were given a brief survey asking about their results (was the code CT or not, etc.), their experience with the tool during the task, and issues they encountered. After the last of the repair tasks, we gave participants a longer exit survey, which included the System Usability Scale [24] and questions regarding their overall experience with the tool. Participants were asked, e.g., whether they trust their tool to give them correct results and what their biggest problem was while using it. A similar survey was included after each audit task.

Instrument Development. Our group of authors consisted of experts in cryptographic engineering, side-channel attacks, and CT tool developers, as well as one human-factors researcher. We based the study development on our usability criteria and related features. We also let our experience with the development of cryptographic libraries and CT verification tools (as authors as well as users) influence the study design. The human-factors researcher introduced and facilitated the use of human-factors research methodology to better explore the identified usability criteria. In particular, the human-factors researcher explained methods when appropriate, facilitated discussions and helped the team to develop the study, pilot it, gather feedback, and evaluate the results.

Pre-Testing. Three co-authors dry-ran the study, followed by one student from the targeted population. Using their feedback we updated, expanded, and clarified the study.

Time Frame. Every participant had a recommended and self-enforced time limit of 8 hours to work on each part of the study (repair and audit, thus a total of 16 hours), within a soft frame of 2 weeks. We allowed extension of the 2-week time frame. Participants, although encouraged to fully use their time, were allowed to hand in their results earlier.

Repair Task Details. The first four of our tasks demonstrate the main points of the CT criterion: Secret-dependent branching and secret-dependent memory access. Repair tasks 01 and 02 are non-CT and CT examples of a selection based on a secret value, once with a branch and once with an arithmetic transformation like the one presented by Schwabe at ShmooCon 2015 [96]. Repair tasks 03 and 04 are likewise memory accesses depending on a secret value or boolean arithmetic for selecting a value loaded from all addresses without depending on the secret for the load address.

Repair task 05 introduces the use of a C programming language standard library function, memcmp, which is non-CT, to compare secret values. Tools which depend on static binaries and cannot inspect dynamically loaded libraries—which are the majority of deployed software today—are expected to fail here and show no CT violation. Repair task 06 includes a system call to read random numbers. System calls are on most operating systems implemented in a way that cannot be seen from user space, the memory area that is analyzable to most CT analysis tools. The tools can work around this, for example by recognizing a set of known system calls and their expected behavior. This task greatly differs from previous tasks, as it does not include secrets, but only checks for support analyzing this code. Task 07 starts to build up problems toward a harder criterion than CT - probabilistic CT, which is a criterion for functions that behave CT by default except for a subset of cases. Indeed, the program reads a random number like in task 06, but in 1 out of 256 cases, it will perform secret-dependent branching like in task 03. This may sound easy to spot manually by most users, but statistics-based CT tools were expected to underperform on this task. Task 08 introduces a different, and on first sight trivial problem: the same function is called, but in two branches based on the value of a secret. This may seem to be CT, but in practice a
compiler may transform this into assembly code that does not branch on the secret, even though in the given source code the CT criterion is violated. The intent behind this task is to see if the abstraction level of a tool, whether it works on binaries or instrumented source code, has a measurable impact on the success of participants. Task 09 makes the compiler transform impossible by changing the branching structure, passing a secret variable as a function input. The called function just returns a constant, which makes the whole program CT. Finally, task 10 is distinguished from previous tasks. It is formally non-CT and can be repaired in two non-obvious ways: users can either make it CT, but only probabilistically correct, or correct, but only probabilistically CT. With this last task, we wanted to see how participants pick up on less trivial code, inspired by techniques used in some cryptographic algorithms recently standardized by NIST, such as Kyber [10, 18] and Dilithium [43].

3. Study Setup. Our tool selection is explained in Section 3.2. Note that one of the included tools (BINSEC/REL) did receive a substantial update during the study, that we did not include as not to invalidate our study.

In order to have similar working environments, we deployed one VM for each tool, and gave SSH access to the participants. Each participant had restricted access to their home directory, with all necessary material (such as instructions and source code of the task and the library) available. For each resource, a clean copy was available as read-only in case they needed a fresh start. We decided to pre-install the tools on the VMs. The reasoning for that choice is twofold. First and foremost, most tools are the outcome of academic research, and served the purpose of demonstrating new techniques and approaches, without aiming for maintainability. Hence, some tools are not maintained, and rely on specific version of dependencies that are outdated and deprecated. This can make the installation particularly complex and time consuming, especially on recent systems. Second, given that participants using the same tools were co-located on a VM, we could deploy the tool globally to ensure a functional setup, and avoid unintentional corruption of the tool by participants.

As an effort to make the first step easier, we implemented installation scripts for each tool present in our study—for possible difficulties in the installation phase, see Reynolds et al. [92]. We made them publicly available (see Footnote 3), along with the repair tasks and a small functional tutorial we provided to the participants. We hope this can prove useful, and motivate tool developers to do the same.

To make sure the participants have something to find in the audit tasks, we needed to include libraries that had problems with CT-ness, therefore we chose the following: two of them—OpenSSL and GNUTLS/Nettle—were chosen because they are in ubiquitous use in open-source software projects. The other—mbedtls—was chosen because it was common and is targeted more for use on embedded devices. Other libraries like BearSSL were not included due to fewer documented CT issues and fewer prior audits of those libraries compared to the first two. We specifically audited the libraries ourselves, first, to see if participants can meaningfully find code that is non-CT in those libraries, either by looking at public documentation and then verifying with a CT verification tool, or direct analysis. All three chosen libraries document which parts of their code bases are not expected to be CT, so our participants could be expected to find them.

4. Coding and Analysis. All qualitative coding and data analysis were done by multiple researchers from a set of four, each coding part done by at least two, from diverse backgrounds and views. All of those researchers were familiar with CT verification, open-source and cryptographic code development practices, while two researchers had additional experience with human factors research with developers. We followed the process for thematic analysis [22]. The four coders familiarized themselves with the free-text answers in their part of the analysis, adding annotations and developing themes as well as codebooks.

Codebooks were first developed deductively based on the questions on each subtask, then changed inductively. The codebooks were iteratively changed while extracting themes from the free-text answers. Coders discussed until agreement was reached to make unanimous decisions; we therefore do not calculate inter-coder reliability [80]. The codebooks codify experiences—good or bad—as well as misconceptions, insecurities, and wishes encountered during the study’s surveys.

5. Data Collection and Ethics. Our invitations were sent to participants of thematically fitting courses of five participating universities. We invited students by emailing them individually. During and after the study they could opt-out of participation. We only linked participants names to results for payment, not during analysis and not by members of the research team who had prior contact to those students. We keep the participant responses as confidential as possible and do not link quotes to them by name, only by pseudonyms. The study protocol and consent forms (for study participation and surveys) were approved by our lead institution’s data protection officer and ethics board, who determined that the study poses minimal risk. Identifying data of the participants, like names, email addresses, and payment information, were stored separately from study data, and were only used to contact the participants; we did not retain any identifying data in excess of following laws.

6. Data cleaning & Presentation. From the 74 students we invited, 31 started our study, of which we were able to use the results of 24 participants. We only evaluate the results of participants who finished a meaningful part of our study and
compared results with and without familiarization with each tool on each library to offset possibly bad pairings, but did not find any meaningful differences between the two groups. As for the 7 incomplete results, we were not meaningfully able to incorporate them in most of the statistics—to not over represent results from simpler tasks—but we used partial results that were complete in appropriate sections.

Participants were paid for and expected to spend two days of eight hours each on the study, leaving rich free text comments in the surveys as well as comments in source code of their task solutions. We received mixed feedback, from disillusioned responses to high interest in further research on CT verification and coding practice. Generally, the feedback to our study was positive, even when the comments about the experience with some tasks were less so.

7. Familiarization. By design of our study, we set our participants up for familiarization with one tool each, then we ask to analyze a common real-world cryptographic library with the same tool. The repair tasks during the familiarization procedure were optimized for familiarization with the tool from simple examples to simplified current research problems.

8. Limitations. Survivorship bias [76] might taint the results, due to the study not reporting all the results of participants which dropped out. Selection bias due to comparatively high requirements in recruiting for the study as well as selective perception due to recruiting from student population who is accustomed to writing exams and tests might both also be relevant, but are both similar to the population which might use one of the CT tools. Participants may have reported more familiarity with the subject matter than they actually had, but due to our recruiting criteria, this was limited to a minimum actually necessary for participation. Our study may also suffer from the typical effects of fatigue in participating in a study, frustrations, and, of course, took place during the later years of the COVID-19 pandemic. Finally, our low participant numbers (due to the significant time investment and prerequisites) does not allow for statistical inference; we report numbers to highlight trends and/or outstanding observations.

Problem Fixing. When participants marked a repair task as already constant-time they were not asked to fix the code.

Library Selection. The projects we included for the audit task represents a selection and are not representative of all open-source cryptographic libraries. We are aware that other libraries might lead to different usability results.

Unknown Code. Our participants were not familiar with the cryptographic libraries used in this study. Annotating and custom-building are likely to be different when analyzing a project the participants are familiar with. Developers might achieve different results if they have a rough overview of the code base. We expect completing the repair tasks to be easier to our participants than the open-ended audit tasks.

VMs, Tutorials, and Examples. By including ready-made virtual machines with each installed CT tool, combined with layered introduction of tasks and documentation, not restricting online documentation and providing some as a backup ourselves, we provided our participants with a best-case scenario to learn how to work with each of the tools. Participants could approach the study as they saw fit, while being able to adapt example solutions and their own prior solutions to everything after a first introduction to the base cases for CT programming practice in minimal examples (tasks 01 to 04). This was a trade-off to gather more data about all CT analysis steps, not being stuck over installation or finding documentation. Nevertheless, this means that our participants had an easier task with the tools than users would have in real world.

## 5 Results

### Table 1. Average and standard deviation of System Usability Scale scores from exit surveys after repair and audit tasks.

<table>
<thead>
<tr>
<th>Tool (Tech., Guar.)</th>
<th>Repair</th>
<th>Audit 1</th>
<th>Audit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINSEC/REL (Sy, ●)</td>
<td>33.5 (3.8)</td>
<td>38.7 (11.8)</td>
<td>45.6 (7.2)</td>
</tr>
<tr>
<td>cterror (F, ●)</td>
<td>30.6 (18.4)</td>
<td>34.4 (8.5)</td>
<td>31.5 (14.8)</td>
</tr>
<tr>
<td>dudect (St, ○)</td>
<td>53.1 (29.1)</td>
<td>65 (5.9)</td>
<td>59.4 (23.8)</td>
</tr>
<tr>
<td>haybale-pitchfork (Sy, ●)</td>
<td>64.4 (6.6)</td>
<td>52.5 (13.7)</td>
<td>50.6 (26.6)</td>
</tr>
<tr>
<td>MemSan (Dy, ●)</td>
<td>49.5 (20.3)</td>
<td>41.3 (22)</td>
<td>49.4 (20.1)</td>
</tr>
<tr>
<td>timecop (Dy, ●)</td>
<td>71.2 (6)</td>
<td>69.4 (10.3)</td>
<td>70.6 (24.1)</td>
</tr>
</tbody>
</table>

Table 1 showcases the System Usability Scale (SUS) scores [24] for each tool on both repair and audit tasks. The SUS is supposed to give a quick overview of a tool’s overall usability; a score above 68 would be “above average” across software types. From the scores presented, usability remains fairly consistent between repair and audit, with notable exceptions for haybale-pitchfork, which had a noticeable dip during audits, and dudect, which exhibited enhanced usability in the audit tasks. Among the tools, timecop has the highest and most consistent score, suggesting superior usability. In contrast, cterror and BINSEC/REL emerge as the least usable. For the correctness of solving the repair tasks, see Table 2.

Through thematic analysis of feedback during repair tasks, we identified common usability issues with the tools. Feedback points, categorized according to our codebooks, often overlapped, except for the “no issue” category. The distribution, depicted in Fig. 2, gave insights into tool perceptions and task challenges.

In Section 6, we delve into the diverse factors impacting usability, as organized by the criteria introduced in Section 3.1, and report on participants’ confidence in their results. Each criterion corresponds to a step in detecting/fixing CT violations, and each subsequent step depends on the success of its predecessor. Those who encountered initial setbacks of-
ten did not report in the later stages. This was particularly pronounced when auditing real-world software libraries.

Our findings spotlight factors affecting the tools’ usability: Clear and intuitive outputs stood out as extremely important, and a lack of beginner-friendly documentation emerged as a recurrent issue. Though well-structured documentation is invaluable during the familiarization phase, participants reported distinct challenges as they delved deeper into the tools, but mixed with positive feedback as well. This disparity became evident when contrasting feedback between standard textbook examples and real-world audits, emphasizing different stages of tool assimilation.

Although the participants were equipped with the tools for the tasks, their installation and setup experiences could not be included in our data as we set up the tools for them.

1. Familiarization. During the first task, the main issue reported was unclear and non-user-friendly documentation, with 16 complaints (18 overall). Although the tools had associated academic papers, participants felt these didn’t serve as effective documentation. They particularly missed step-by-step setup and results interpretation examples. “[T]he documentation about every command doesn’t exist or I didn’t find them. Maybe a beginner-friendly aspect of the tool would have been good for me to start.” (P19)

Notably, participants had no complaints about cverif documentation—possibly due to its basic user interface—but most of them faced issues with its operational aspects until they consulted our tutorial.

Despite the issues, our study also highlighted successes in the familiarization phase. Concise, beginner-focused documentation was identified as a significant upside in enhancing user engagement. The turnaround is likely a direct result of the tutorial we provide upon the completion (or non-completion) of the first task.

None of the participants managed to solve the second task using cverif, and all expressed complaints about the output. After the tutorial and solution were provided, 3 out of the 4 participants were able to solve the subsequent task. This improvement persisted through the remaining tasks and can be attributed largely to the alleviation of difficulties in correctly interpreting the output and running the tool.

The effect was similar with Binsec/Rel. While none of the participants solved the first task, 4 out of 5 successfully solved the second task following the tutorial.

We noticed that our tutorials had a particularly strong impact on the usage of dudect, a tool that elicited the most complaints about lack of documentation. One participant even expressed their appreciation with the following: “Great tutorial about dudect on the previous study page. Why it is not included in the official documentation?” (P31)

Overall, the tutorial was appreciated for every tool in the repair tasks, as suggested by the following quote. “I am just really using the template provided in the tutorial” (P14).

During the audit task, 15 struggled to start using the tool, despite our tutorial. This was especially true for Binsec/REL (3), cverif (5) and haybale-pitchfork (4). Complaints mainly referred to lack of guidance in more complex tool usage, such as hooking functions, or bypassing some tool limitations. Tools with a more straightforward functioning, such as timecop and MemSan, did not suffer from these complaints.

2. Building and Secret Designation. This crucial pre-processing step is fraught with complexity, leading to 30 complaints from the study’s participants during the repair task.

Central to the participants’ challenges was the task of designating the secret within the given code snippets. The complexities arose either from the need to annotate the code, leading to 17 complaints, or the requirement to design a wrapper, which received 7 grievances in total. In particular, the annotation APIs provided by Binsec/REL and cverif were deemed overly complicated. This perspective was substantiated by 7 and 9 complaints, respectively, suggesting poor usability. In contrast, MemSan and timecop offered more streamlined processes, simply enabling users to flag a memory region as secret. The challenge of implementing external wrappers for tools like dudect and haybale-pitchfork was accentuated by insufficient documentation, evidenced by 4 and 3 feedback reports. A unique challenge presented by haybale-pitchfork was its reliance on the Rust language, which impeded 3 par-
Participants. This prerequisite even pushed one to abandon the study. Resistance to continue, even when participants were provided with ready-to-use tools and monetary encouragement, underscores usability concerns for the target user base.

Interestingly, the audit tasks unveiled a new set of foundational hurdles. A seemingly rudimentary step - local library installation - became a roadblock for 13 participants across all tools. While participants found the compilation of minor repair tasks with specified options manageable, the challenge escalated when they had to adapt intricate compilation chains to enable the tool use. In this regard, 16 participants faced hurdles when gearing up the libraries for suitable compilation to enable analysis. The architectural constraints of Binsec/Rel, especially the need to compile libraries for a 32-bit architecture, caused difficulties for 7 participants (given the study reliance on an older tool version). Haybale-pitchfork posed its unique challenge, with 5 participants coping to generate the necessary bitcode of the library. The tools dudect and haybale-pitchfork added another layer of complexity by necessitating external wrappers, proving problematic for 4 and 1 users. The demands of accurate code annotation further intensified the complexities during this phase for 4 participants. This was notably severe for Binsec/Rel users and MemSan, 2 reports each. Overall, 10 participants faced significant hurdles in advancing further in the library audit, and did not manage to run the tool. 6 of them were blocked when using ctverif.

3. Analysis Runtime. In the context of the repair tasks, while many tools were wielded effortlessly on multiple tasks—indicated by the "no issue" category in Fig. 2—both Binsec/Rel and ctverif manifested signs of a higher barrier, even for tasks that appeared superficially straightforward. Specifically, Binsec/Rel was utilized seamlessly on 15 occasions, whereas ctverif demonstrated hassle-free operation only 9 times. We want to highlight the particular difficulty participants faced with Binsec/Rel during the first repair task. Users were presented with a multitude of options, some of which tangential to the main task, leading to 3 complaints.

The audit phase, characterized by the need to analyze larger code bases, brought forth a different set of issues. The time-consuming nature of the analysis was a concern, particularly for haybale-pitchfork and dudect. Analysis processes were identified as overly protracted by 1 and 2 participants respectively. This drawn-out analysis underscored concerns over the efficiency and practicality of these tools in real-world settings.

4. CT Problem Fixing. A preliminary glance at the success metrics in utilizing the tools, referenced in Table 2, exhibited significant disparities among the tools. Some adopted a tool-reliant strategy, while others, having initially engaged with a tool, later pivoted to manual code analysis. Given the easy nature of most tasks, forcing participants to resort to manual analysis is a witness of poor usability. We recorded these events mostly with ctverif, Binsec/Rel and dudect.

Participants unanimously agreed that discerning the leakage and subsequently mitigating it constituted the principal challenges. These were reflected in 77 grievances. The main subset of these, amounting to 51, expressed that after detecting the leakage, the repair process itself posed difficulties. These difficulties could arise from both details of the tasks and participants' limited familiarity with CT programming. The documentation most consulted by participants was related to CT programming methodologies, suggesting that the primary impediment might be their inexperience in this domain rather than difficulties with the tools themselves. We think this inexperience is not an impediment to use the tools, just in fixing more advanced problems in the code. This observation aligns with our expectations given the demographic we recruited for the study.

Tool outputs and how to interpret them emerged as a recurring concern, in 26 documented instances. Participants grappled with either a lack of comprehensive documentation to interpret the output (15 instances) or ambiguous outputs that did not offer a conclusive determination on the code CTness (11 instances). Here, dudect and haybale-pitchfork stood out for their clarity and precision as seen from little complaints in participant feedback. This likely results from tools concluding their analysis with a definitive statement about the status of the analyzed code, whereas other tools tend to provide information about possible issues, which can be confusing for beginners. Binsec/Rel and ctverif gathered criticism for occasional vagueness, with 2 and 9 mentions.

The relatively fewer complaints associated with dudect (3 instances) can likely be attributed to its methodology and careful wording of reports.

Even though we knew of pre-existing CT violations, 12 participants reported to be unable to detect any of them. These observations include use of haybale-pitchfork (4 participants) and Binsec/Rel, dudect, and timecop (2 participants). For both MemSan and ctverif it was reported once.

5. Specialized Output Generation. Participants voiced concerns with the verbosity and confusing nature of elaborate error reports. A segment of the study population—3 participants out of 24—grappled with decoding these verbose outputs during the audit tasks. These participants found it challenging to distinguish actual CT violations amid the warnings, and were overwhelmed by the volume of output. Specifically, participant feedback highlighted ctverif as the most problematic in this regard, accounting for 3 complaints. These complaints were directed toward errors preventing its proper usage, and not CT violations. Timecop had 2 mentions, while other tools, barring haybale-pitchfork, were criticized once each.

6. Reliability / False Positives. Throughout the repair tasks, participants expressed skepticism regarding the tools, registering 18 complaints centered on perceived reliability issues.
Such concerns typically revolved around reports of false positives (recorded 4 times), false negatives (4 times), mistrust in the results (2 instances), or specific tool reasoning limitations (8 times). Among all the tools evaluated, timecop stood out with no reliability complaints. In stark contrast, MemSan found itself at the receiving end of the most criticisms—amounting to 5, predominantly targeting perceived limitations in its analysis. BINSEC/REL follows with the same amount of complaints, but mostly on false positive.

9 participants successfully modified task 10, which was designed to be easily detectable as non-CT yet challenging to fix, in a manner where they discussed their solutions statistically probable CT-ness or correctness compared to the given setting. These successes provide a valuable insight: even when faced with complex tasks, participants can learn and adapt to the nuances of the tools. Due to our provided documentation and the ramp-up of repair task difficulty, we allege that these findings underline the significant role of quality documentation in the tool experience.

Upon transitioning to library audits, participants generally exhibited more restraint in identifying false positives or negatives than in the repair tasks. They mostly expressed low confidence in their analysis, evidenced by 20 self-report on low confidence. “I have very low confidence in these results that must be false due to my usage of the tool.” (P06) This reticence was particularly pronounced for BINSEC/REL and MemSan, which gathered 5 and 4 reports, respectively. The participants detected few to no CT violations while analyzing libraries using haybale-pitchfork. This was reported 4 times. With BINSEC/REL, dudect and timecop, only two such cases were reported, and with MemSan and ctverif only one each. Low detection rates are usually correlated with issues during prior steps of the analysis, whether for preparing the library for audit or for using the tool. We conclude this by looking at inter repair task success rates in the first part of the study.

Despite the grievances recorded, most tools, except for ctverif, proved effective in detecting non-CT code during the audit tasks. dudect emerged as the front runner, recording 6 reports of successful detection, followed by BINSEC/REL and timecop, which facilitated 4 reports of discoveries each. Further, haybale-pitchfork accounted for 2 instances, and MemSan contributed 1 finding. We regard these outcomes as practical successes in identifying potential CT-violating bugs in the analyzed open source production code. We did not report these known and (upstream) documented findings.

### 6 Discussion

Building on the results of our study, we discuss the usability of different tools and make a series of recommendations based on the different stages of usage used in our study.

#### 6.1 Usability vs verification approaches

Our results provide relevant information on the usability of tools relative to the verification approach they use.

Our study suggests that users found dudect intuitive to use. On the other hand, the underlying approach of statistical time measurement demands a strategic minimization of test parts when dealing with large code bases. Interestingly, the technique that lengthened dudect’s processing time might have also contributed to its user-friendliness. The developers of dudect appeared to balance precision with early termination options for less accurate but faster results. Consequently, our participants found the output more intuitive. Whether participants knew they were trading accuracy for speed is unclear. Although it operates as a “black box”, a careful balance of precision, speed, and clarity in dudect made it an effective tool for our participants, as seen from their success rates—as seen in Table 2—and feedback.

Dynamic instrumentation tools often have a tug-of-war between technical efficiency and user experience, posing challenges during the setup phase. MemSan also faced significant trust issues due to perceived unreliability. timecop stood out with its blend of efficiency and user-friendliness. haybale-pitchfork, proficient yet challenging for some users, hinted at possible issues in the prior analysis steps.

Formal analysis tools, namely BINSEC/REL and ctverif, stand out due to their capacity to offer strong guarantees based on rigorous semantics. While robust, the theoretical foundation of these tools can come at a cost in terms of user experience. Specifically, ctverif presented a series of usability hurdles, from its initial installation to operational procedures. Many participants encountered challenges despite being provided with a working installation and sample use cases, leading to less successful task resolutions. In addi-
tion, our participants did not report particularly more trust in cverif’s output, despite the strong guarantees it claims. On the other hand, Binsec/REL demonstrated that it is possible to maintain strong analytical guarantees while ensuring a more straightforward setup and operational process. This contrast between the two tools underscores the significance of balancing analytical capabilities with an intuitive user experience when time efficiency and ease of use are highly valued [61].

Our study offers a nuanced perspective on the usability-efficacy spectrum of different analysis tools. While strong guarantees are a primary concern, the trade-offs with usability can sometimes diminish a tool’s practical application. The findings emphasize the need for tool developers to prioritize both rigorous analysis capabilities and a seamless user experience, ensuring that state-of-the-art tools are not just theoretically sound but also practically adoptable.

### 6.2 Recommendations

We combine the data from our empirical study with expert insights to curate a suite of recommendations. Our observations indicate that the tool usage is divided into multiple stages. Of particular concern, inhibiting complexities at early stages can deter users from progressing.

**Installation.** Many tools have a large number of dependencies and require custom building paths. As a result, installing these tools may be highly challenging in the mid-term, even if all the tool’s dependencies are maintained. From study setup and piloting we extract the following recommendations:

- Reduce and avoid less maintained dependencies.
- Make tools available via package managers.

A more general recommendation would be to embrace the best practices of open-source software development, which has a long, integrated maintenance period and is often available as native packages in software distributions—native packages through distributions also make for discoverable tools.

**Familiarization.** After installation, users may run the tool on common examples, in this case crypto libraries, just to make sure that the tool is indeed running, and without caring for the tool’s results. However, there are many obstacles to such dry runs. This includes, for instance, the need to compile libraries using specific compilation flags, different from the flags used to produce code, or the need to rewrite libraries to overcome limitations in the coverage of the tool. To avoid such situations and to ensure that tools provide adequate support for beginners, we make the following recommendations:

- Provide support for processing inline assembly and vectorized cryptographic code.
- Provide support for processing precompiled code, statically or dynamically linked.
- Provide user-friendly examples amenable to adaptations.
- Design intuitive tutorials catering to novices, and covering all the aspects of tool-usage.

- Prioritize a comprehensive documentation structure, accentuating essentials before delving into details. Make sure that the documentation avoids overly specialized jargon.

The latter recommendations are based on the feedback from the study participants.

**Secret Designation.** Most constant-time tools require users to provide security annotations. The annotations are typically given in the code or through some external wrapper. Moreover, many cryptographic libraries require users to declassify computations, for example to make ciphertexts public. From prior literature on different tool designs and their problem areas, we extract the following recommendations:

- Make annotations simple and external in additional files.
- Provide mechanisms to declare internal secrets [49].
- Provide mechanisms to allow to declassify computations.

**Output generation.** Results of analysis tools must be semantically rich, easy to navigate, and exploitable in a broader setting. Based on our interpretation of the results of the study, and our expertise, we make the following recommendations:

- Provide output that is readily understandable by users, including origin of leakage.
- Offer the possibility to report all leakage violations at once.
- Deduplicate findings in order to avoid repeating violations.
- Offer export formats for integration with bug-tracking tools.
- Have a delta mode for CI.

**Analysis Runtime.** For integration into users’ workflows or CI, analysis should be possible in reasonable time—we think a few minutes are fine even for interactive use, but hours or longer are not. From Jancar et al. [61] as well as our own study participants’ feedback and the CPU utilization of our study setup, we make the following recommendations:

- Use progress indicators (progress bar or logs) to ensure the user understands the tool is not stalled.
- If applicable, leverage multiple CPU cores for large tasks.

### 7 Conclusion

We collected data from 24 participants using 6 CT analysis tools to analyze small tasks and audit 3 open-source cryptographic libraries that are documented not to be fully CT. Our broad conclusion is that CT analysis tools have usability shortcomings that prevent them from being integrated into developers’ workflows. Although our analysis focused on CT tools, we believe that many of our findings also apply to tools for analyzing microarchitectural side-channels. We believe the community should address these shortcomings by focusing on a handful of easy-to-use and maintained tools that go beyond the CT leakage model and cover a broad range of leakage models.
Acknowledgements

This work was supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE). J. Jancar was supported by MV AI-SecTools (VJ02010010) project and by Red Hat Czech. Daniel De Almeida Braga was funded by the Direction Générale de l’Armement (Pôle de Recherche CYBER). We thank Anna Lena Rotthaler, our anonymous reviewers, and our shepherd for helpful comments. We thank all participants for their time and effort spent on this research.

References


Summary of CT analysis tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Target</th>
<th>Tech.</th>
<th>Guar.</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABPV13 [5]</td>
<td>C</td>
<td>Fo</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>ABSynthe [51]</td>
<td>Leakage</td>
<td>Dyn</td>
<td>■</td>
<td>Github</td>
</tr>
<tr>
<td>ANABLEPS [116]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>BINSRC/REL [38]</td>
<td>Binary</td>
<td>Sym</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>Blazer [6]</td>
<td>Java</td>
<td>Fo</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>BPT17 [16]</td>
<td>C</td>
<td>Sym</td>
<td>●</td>
<td>irisa.fr</td>
</tr>
<tr>
<td>CacheAudit [41]</td>
<td>Binary</td>
<td>Fo</td>
<td>■</td>
<td>Github</td>
</tr>
<tr>
<td>CacheAudit2 [42]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>Cached [115]</td>
<td>Trace</td>
<td>Sym</td>
<td>○</td>
<td>no</td>
</tr>
<tr>
<td>CacheFix [34]</td>
<td>Trace</td>
<td>Sym</td>
<td>○</td>
<td>BitBucket</td>
</tr>
<tr>
<td>CacheQL [127]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>CacheS [114]</td>
<td>Binary</td>
<td>Fo</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>CANAL [100]</td>
<td>LLVM</td>
<td>Fo</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>Cache Templates [53]</td>
<td>Binary</td>
<td>Stat</td>
<td>■</td>
<td>Github</td>
</tr>
<tr>
<td>CaSym [25]</td>
<td>LLVM</td>
<td>Sym</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>CaType [63]</td>
<td>Binary</td>
<td>Fo</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>CHALICE [33]</td>
<td>LLVM</td>
<td>Sym</td>
<td>○</td>
<td>BitBucket</td>
</tr>
<tr>
<td>COCO-CHANNEL [23]</td>
<td>Java</td>
<td>Sym</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>Constantine [17]</td>
<td>LLVM</td>
<td>Dyn</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>ctgrind [75]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>ct-fuzz [56]</td>
<td>LLVM</td>
<td>Dyn</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>CT-WASM [118]</td>
<td>WASM</td>
<td>Fo</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>DATA [119,120]</td>
<td>Binary</td>
<td>Dy</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>DifFuzz [85]</td>
<td>Java</td>
<td>Dyn</td>
<td>○</td>
<td>no</td>
</tr>
<tr>
<td>dudect [91]</td>
<td>Binary</td>
<td>Stat</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>ENCIDER [126]</td>
<td>LLVM</td>
<td>Sym</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>ENCoVer [8]</td>
<td>Java</td>
<td>Fo</td>
<td>●</td>
<td>kth.se</td>
</tr>
<tr>
<td>FlowTracker [93]</td>
<td>LLVM</td>
<td>Fo</td>
<td>●</td>
<td>ufmng.br</td>
</tr>
<tr>
<td>haybale-pitchfork [106]</td>
<td>LLVM</td>
<td>Sym</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>KMO12 [69]</td>
<td>Binary</td>
<td>Fo</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>Manifold [128]</td>
<td>Binary</td>
<td>Stat</td>
<td>○</td>
<td>Zenodo</td>
</tr>
<tr>
<td>MemSan [59]</td>
<td>LLVM</td>
<td>Dyn</td>
<td>●</td>
<td>llvm.org</td>
</tr>
<tr>
<td>MicroWalk [121]</td>
<td>Binary</td>
<td>Dyn</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>MicroWalk-CL [122]</td>
<td>Binary</td>
<td>Dyn</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>PinCEC [59]</td>
<td>Binary</td>
<td>Dyn</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>Pitchfork-angr [107]</td>
<td>Binary</td>
<td>Sym</td>
<td>○</td>
<td>Github</td>
</tr>
<tr>
<td>SC-Eliminator [123]</td>
<td>LLVM</td>
<td>Fo</td>
<td>●</td>
<td>Zenodo</td>
</tr>
<tr>
<td>Shin et al. [97]</td>
<td>Binary</td>
<td>Stat</td>
<td>○</td>
<td>no</td>
</tr>
<tr>
<td>SideTrail [7]</td>
<td>LLVM</td>
<td>Fo</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>STACCO [124]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>no</td>
</tr>
<tr>
<td>STANAlyzer [95]</td>
<td>C</td>
<td>Fo</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>Themis [35]</td>
<td>Java</td>
<td>Fo</td>
<td>●</td>
<td>Github</td>
</tr>
<tr>
<td>timecop [84]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>blog</td>
</tr>
<tr>
<td>tis-ct [37]</td>
<td>C</td>
<td>Sym</td>
<td>●</td>
<td>no</td>
</tr>
<tr>
<td>TLSfuzzer [65]</td>
<td>Network</td>
<td>Stat</td>
<td>■</td>
<td>no</td>
</tr>
<tr>
<td>TriggerFlow [52]</td>
<td>Binary</td>
<td>Dyn</td>
<td>○</td>
<td>Gtlab</td>
</tr>
<tr>
<td>VirtualCert [11]</td>
<td>x86</td>
<td>Fo</td>
<td>●</td>
<td>edu.yu</td>
</tr>
</tbody>
</table>


Technique: Sym—Symbolic, Stat—Statistics, Dyn—Dynamic, Fo—Formal, —also performs code transformation/synthesis

Guarantees: ●—sound, ○—sound with restrictions, ■—no guarantee, —other property

Table 3. Classification of CT tools.