Automated Side-Channel Analysis of Cryptographic Protocol Implementations

Faezeh Nasrabadi
CISPA Helmholtz Center for Information Security & Saarland University
faezeh.nasrabadi@cispa.de

Robert künnemann
CISPA Helmholtz Center for Information Security
robert.kuennemann@cispa.de

Hamed Nemati
KTH Royal Institute of Technology
hnnemati@kth.se

Abstract

Formal verification of cryptographic protocols typically relies on symbolic models that abstract away compiled code and microarchitectural side channels, leaving a gap between verified specifications and deployed executables. We present a toolchain that extracts protocol-relevant models from real binaries and analyzes them under explicit leakage contracts for constant-time and Spectre-PHT-style speculative observations. Starting from a selected binary region, we lift machine code to an intermediate representation, instrument it with leakage contracts, symbolically execute it to obtain event/observation traces, and translate these traces into SAPIC$^+$ for analysis with TAMARIN, PROVERIF, and DEEPSEC.

As case studies, we extract models of WhatsApp Desktop's session-management and double-ratchet components from its binary and analyze forward secrecy and post-compromise security under a state-cloning compromise. For side-channel analysis, we study the Basic Access Control (BAC) protocol used in e-passports and WhatsApp's session establishment. Under our observation models, we identify an instruction-cache side channel in WhatsApp Desktop enabling social-graph inference, and we reproduce known unlinkability issues in BAC under microarchitectural observations.

CCS Concepts

• **Security and privacy → Formal methods and theory of security**.

Keywords

Formal Analysis, Crypto. Protocols, Side Channel, Binary Analysis

ACM Reference Format:
Faezeh Nasrabadi, Robert künnemann, and Hamed Nemati. 2026. Automated Side-Channel Analysis of Cryptographic Protocol Implementations. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, Hague, Netherlands.*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

CCS '26, Hague, Netherlands.

ACM ISBN XXX
https://doi.org/10.1145/XXXXXX.XXXXXX

1 Introduction

Cryptographic protocols form the backbone of modern digital security, protecting sensitive data across online interactions. Yet their design and implementation remain prone to subtle errors both at the specification level (e.g., the POODLE attack on SSL version 3.0, which exploited SSL's use of CBC-mode encryption with predictable padding [27]) and at the implementation level (e.g., Heartbleed, `CVE-2014-0160`). Formal methods can provide a systematic framework for finding attacks and even ensuring the absence of (defined classes of) them. But to find attacks on both levels, we need models that capture the behavior of the compiled machine code, which is what ultimately runs on hardware. Overcoming this *gap* promises to increase the trustworthiness of analyses, but also seems necessary considering the rising complexity of modern protocols, which often involve various sub-protocols like session management and key exchange. It also offers an opportunity to further investigate and identify attacks resulting from hardware interaction. In particular, *side-channel attacks* are known to leak sensitive information.

Recently, CRYPTOBAP [73, 74] sought to bridge this gap by verifying cryptographic protocols at the machine-code level; while binary-analysis platforms such as BINSEC [46] and BAP [29] offer powerful program-analysis capabilities, they are insufficient for such analyses, as they lack any mechanism to semantically reconstruct protocol logic from binary. Being in its early development stages, CRYPTOBAP suffers from limited scalability, restricting analysis to small or medium-sized protocols. Its limitation mostly comes from the computational overhead of techniques like symbolic execution on large code bases and the difficulty of modeling dynamic protocol behaviors (e.g., session resumption, key rotation) in automated verifiers. Moreover, CRYPTOBAP focuses mostly on trace properties in the Dolev–Yao model (e.g., secrecy, authentication) and does not account for hardware-induced side-channel vulnerabilities. This oversight raises concerns, as side-channel leaks in protocol implementations can inadvertently expose secrets; however, so far side channels were not taken into account when analyzing protocol *implementations*. Traditional side-channel detection tools [34, 43, 53, 76, 89], which target cryptographic primitives or library code, fail to account for protocol-specific interactions that amplify leakage, such as timing variations that can happen during session establishment, error handling, order of operations, or memory-access patterns during key derivation.

In this paper, we present a methodology for analyzing cryptographic protocol implementations, addressing both classical security properties and side-channel resilience. Our approach extends

CryptoBap [73, 74] with observation models [30, 76] (a.k.a., leakage contracts [53]) that enable automated analysis of real-world protocols against side-channel attacks. Building on CryptoBap's core methodology—transpiling assembly code into an intermediate representation for symbolic execution and model extraction—we extract protocols' Sapic$^+$ model directly from their binary snippet that can be analyzed by DeepSec for side channel leak and by Tamarin/ProVerif for reachability properties.

Threat model and scope. We consider an active network attacker in the Dolev–Yao model and a co-resident attacker (process) that can observe microarchitectural effects captured by our leakage contracts (e.g., instruction- or data-cache Prime+Probe measurements). For speculative leakage, we adopt a Spectre-PHT-style model in which conditional branches may be transiently mispredicted and wrong-path memory accesses may become observable through side channels. Our goal is *not* to model all microarchitectural channels, but to lift well-scoped observation models to protocol verification and to expose security violations that arise from control flow divergence and memory-access behavior.

Our observation models can, in principle, capture "classical" constant-time violations such as secret-dependent table lookups (e.g., in an AES implementation) because secret-dependent load addresses become observable. However, we abstract crypto library calls to focus on protocol logic and on protocol-level leakages.

As a case study, we conduct the first formal verification of the WhatsApp Desktop binary by extracting a model of its session management protocol (Sesame) and double ratchet mechanism. WhatsApp is the most widely used messaging platform, with over 3 billion users across the globe [3]. While Sesame has been formally analyzed at the specification level [41], no prior work extracted and verified the Sesame logic as *implemented* by WhatsApp's client. Given that WhatsApp is closed-source and similar systems have historically shown vulnerabilities (e.g., EternalBlue, `CVE-2017-0144`), this lack of implementation-level analysis is concerning.

For large applications like WhatsApp, considering all memory operations and function calls yields a large extracted model. Even with simplification, state-of-the-art protocol verifiers still fail to terminate when analyzing models of this scale. For reachability-style properties on WhatsApp (e.g., forward secrecy and post-compromise security), we therefore rely on CryptoBap's original extraction pipeline, which abstracts memory effects and cryptographic primitives in a way that is sound for finding trace-property violations under its assumptions. For side-channel and privacy analyses, we apply our observation-aware extraction to smaller, protocol-relevant components where memory- and control-flow observations are essential and still scalable.

Using our models, we (i) prove forward secrecy of WhatsApp's session establishment and message exchange, (ii) confirm that a clone attacker can break post-compromise security—as previously identified for Signal application [41]—and (iii) identify a novel privacy attack that leverages instruction-cache leakage during session establishment to infer whether two users have been in contact. To summarize, our contributions are as follows:

(1) We extend CryptoBap with leakage contracts and a model extraction path to DeepSec to enable automated reasoning about protocol security in the presence of microarchitectural side channels.
(2) We provide the first formal model of WhatsApp's implementation of session management and double ratchet component extracted from its binary, and use Tamarin/ProVerif to prove forward secrecy and to confirm a post-compromise security break against a clone attacker.
(3) Applying our side-channel-aware analysis to WhatsApp session establishment, we find a privacy leak: a side-channel attacker can distinguish first-time from existing contacts, enabling them to infer the victim's social graph. We also confirm the known unlinkability break in BAC.

$$
\begin{aligned}
\mathbf{P} \in \mathbf{prog} &:= \mathbf{block}^* \\
\mathbf{block} &:= (\mathbf{v}, \mathbf{stmt}^*) \\
\mathbf{v} \in \mathbf{Bval} &:= \mathit{string} \mid \mathit{int} \\
\mathbf{stmt} &:= \mathbf{halt} \mid \mathbf{jmp}(\mathbf{e}) \mid \mathbf{cjmp}(\mathbf{e}, \mathbf{e}, \mathbf{e}) \\
&\quad \mid \mathbf{assign}(\mathit{string}, \mathbf{e}) \mid \mathbf{assert}(\mathbf{e}) \\
\mathbf{e} \in \mathbf{Bexp} &:= \mathbf{v} \mid \diamond_{\mathbf{u}}\ \mathbf{e} \mid \mathbf{e}\ \diamond_{\mathbf{b}}\ \mathbf{e} \mid \mathbf{var}\ \mathit{string} \mid \mathbf{load}(\mathbf{e}, \mathbf{e}, \mathit{int}) \\
&\quad \mid \mathbf{store}(\mathbf{e}, \mathbf{e}, \mathbf{e}, \mathit{int});\ \mid \mathbf{ifthenelse}(\mathbf{e}, \mathbf{e}, \mathbf{e})
\end{aligned}
$$

Figure 1: BIR's syntax

Responsible disclosure: The vulnerabilities found in WhatsApp Desktop were responsibly disclosed to Meta in March 2025. Meta confirmed that the application is vulnerable to identified attacks.

2 Background

We build on CryptoBap [73, 74], a binary analysis framework that extends security protocol verification to machine code to eliminate the need to trust compilers. CryptoBap extends the HolBA framework [62] to verify ARMv8 and RISC-V machine code crypto protocols. It achieves this by extracting a formal model of the protocol under analysis, which can then be translated into models suitable for automated verification using ProVerif, CryptoVerif, Tamarin, and DeepSec. In this section, we provide an overview of the CryptoBap structure and introduce the necessary preliminaries to understand the contributions of this paper.

2.1 HolBA Framework

HolBA [62] is a library for binary analysis based on the HOL4 theorem prover [45] and the L3 specification language [49]. HolBA achieves this by transpiling binary code into the Binary Intermediate Representation (BIR [1]), a simple, architecture-agnostic language designed to facilitate binary analysis and tool development. To ensure soundness, the transpilation process is verified to preserve the semantics of the input machine code.

Figure 1 depicts BIR's syntax. A BIR program P consists of uniquely labeled blocks, with each block containing a sequence of statements. Labels correspond to specific locations in the program and are commonly used as the target for jump instructions. BIR

[1]We use colors for different languages: RoyalBlue, math bold for BIR and SBIR, and RedOrange, sans serif for Sapic$^+$. Common elements use *black italics*.

$\langle P, Q \rangle$ ::=

0	| !P
| in(x); P	| P | Q
| out(x); P	| new n; P
| if ϕ then P else Q	| event e; P
| let $t_1 = t_2$ in P else Q	

Figure 2: A fragment of the syntax of Sapic$^+$ process calculus. In this figure, $e, t_1, t_2 \in \mathcal{T}$, $n \in \mathcal{N}_{\mathtt{priv}}$, $x \in \mathcal{V}$.

statements include (a) **assign**, to assign a **BIR** expression to a variable, (b) jumps (i.e., **jmp** or **cjmp**), (c) **halt**, which serves as the termination instruction, and (d) **assert**, which evaluates a boolean expression and terminates execution if the assertion fails. Expressions in **BIR** include constants, variables, conditionals (i.e. **ifthenelse**), arithmetic operations, denoted by $\diamond_b$ for binary and $\diamond_u$ for unary, as well as memory operations such as **load** and **store**.

HolBA provides a proof-producing symbolic execution for **BIR** [61] which CryptoBap [73, 74] uses in its pipeline. This symbolic execution formalizes the symbolic generalization of **BIR** (called **SBIR**) to explore all execution paths of the program. The symbolic semantics align with concrete semantics, enabling guided execution that maintains a set of reachable states arising from an initial symbolic state. HolBA's symbolic execution allows for verifying functional correctness, but not (directly) protocol security, as it lacks a suitable attacker model and concurrent behavior. CryptoBap bridges this gap by extracting formal models of the protocols from their implementations. This model is then used to analyze security properties using external protocol verifiers. CryptoBap extracts formal models of protocols into two distinct modified versions of the applied π-calculus: one suitable for automated verification using ProVerif and CryptoVerif, proposed in [73], and another for automated verification with ProVerif, Tamarin, and DeepSec, utilizing the Sapic$^+$ toolchain introduced in [74]. Since our goal is to check *trace equivalence* [67], we have chosen to extract Sapic$^+$ models, as Sapic$^+$ backends support equivalence properties.

2.2 Sapic$^+$ & DeepSec

The Dolev–Yao model includes an attacker that exploits logical flaws in a protocol, but cannot compromise cryptographic primitives [47]. In this model, the cryptographic primitives are considered perfect—for instance, guessing a key is impossible. A specific set of rules defines the abstract manipulation of messages, while other alterations are not permitted.

In a protocol execution, messages are represented as terms. High-entropy values are modeled by constants derived from an infinite set of names $\mathcal{N}$ classified into public names $\mathcal{N}_{\mathsf{pub}}$ (available to attackers) and secret names $\mathcal{N}_{\mathtt{priv}}$. Terms denoted as $\mathcal{T}$ are constructed using names derived from $\mathcal{N}$, variables sourced from a variable set $\mathcal{V}$, and by applying function symbols from $\mathcal{F}$ on terms.

Sapic$^+$ [36] is a dialect of applied π-calculus that provides a language that soundly translates to Tamarin [68], ProVerif [26] and DeepSec [37]. Sapic$^+$ enhances Sapic [59] by introducing destructors and let bindings with pattern matching and else branches. A fragment of Sapic$^+$'s syntax is shown in Figure 2. The in and out constructs are responsible for receiving and outputting messages through a channel visible to the attacker. The event construct is used to raise events that pertain to the reachability properties of the model. Furthermore, the new construct enables the generation of new values. Conditionals are described by first-order formulae ϕ over equalities on terms, possibly containing variable quantifiers, as in [59]. The Sapic$^+$ syntax features *stateful* processes that modify globally shared states, which are excluded in Figure 2 for simplicity.

Sapic$^+$ facilitates the analysis of equivalence properties through its backends, specifically DeepSec, a specialized tool designed for this purpose. DeepSec focuses on indistinguishability properties, particularly trace equivalence. It employs a language similar to ProVerif but without the "!" operator for unbounded replication, as it supports only bounded verification. Unlike ProVerif, DeepSec provides a decision procedure that guarantees termination, given sufficient resources. As a result, DeepSec can effectively check trace equivalence in cases where ProVerif fails to terminate, though it requires bounding the number of replications.

Backend protocol verifiers. In our pipeline, Sapic$^+$ serves as a front-end that can be discharged to different backends depending on the property of interest and the required level of automation. ProVerif is typically effective for reachability properties (e.g., secrecy and authentication) and supports unbounded replication, but it may not terminate on complex models. Tamarin supports richer equational theories and an interactive proof mode to guide proof search with lemmas and proof hints. DeepSec provides a terminating decision procedure for bounded verification of process equivalences such as trace equivalence, which we use to encode privacy properties (e.g., unlinkability [11]) and conditional non-interference [52] in the presence of side-channel observations.

2.3 Side Channels & Observational Models

Resource sharing is inevitable in computing due to limitations in available resources. However, if not done carefully, it can introduce unintended information flow channels, also known as side channels. These channels can potentially be exploited by a malicious process to exfiltrate secret information from trusted ones.

Attacks that exploit the data and instruction caches are among the most commonly used side-channel attacks [7, 77, 87, 88]. One widely used technique for extracting information via caches is known as Prime+Probe [82]. In an instruction-cache attack using this technique, first, the attacker **primes** the cache by filling it with their own instructions. Then, while the victim executes, some of the attacker's cached instructions may be evicted. Finally, the attacker **probes** the cache by *measuring access times* to their instructions to detect evictions that reveal the victim's execution behavior.

The number of attack techniques exploiting microarchitectural features, like caches, to leak secret data continues unabated. Consequently, the study of information flow analysis techniques to ensure the absence of information leakages due to side channels is a topic of increasing relevance. A formal model of side channels is essential for such an analysis. However, explicitly modeling all the intricate features of modern processors—like cache hierarchies, replacement policies, and memory interactions—is almost infeasible due to their complexity and because many microarchitectural details are not publicly available. To address this challenge, *abstract observational*

models [30, 76] (a.k.a., *leakage contracts* [53]) provide an alternative by overapproximating an attacker's capabilities.

An observation model $\mathcal{M}$ extends the abstract representations of a processor's operational semantics by a set of system states S, a set of possible attacker observations O and a labeled transition relation $\rightarrow_m \subseteq S \times O \times S$ indexed with the execution mode $m \in \{r, t\}$. When $m = r$, we mean that the processor executes at the software-visible ISA level using a sequential transition system, while with $m = t$, we denote the transition relation of some target microarchitecture where the information flow may be affected by optimizations such as out-of-order or speculative execution. Essentially, observations define which parts of the processor state influence the side channel at each transition. This enables information flow analysis without requiring us to know the exact microarchitectural behavior.

The primary property to formalize the absence of microarchitectural leakages due to side channels is *conditional non-interference* [52]. Let $s \in S$ be a system state, including microarchitectural components like caches, $traces : T \mapsto 2^O$ be a function to extract the sequence of observations from a given execution trace $\tau \in T$, and $s \sim_{\mathcal{M}} s'$ is the state's indistinguishability relation w.r.t. the model $\mathcal{M}$. Then we say:

Definition 1 (Conditional non-interference (CNI))**.** *A system is* conditionally non-interferent *if for all indistinguishable initial states s and s' (i.e., $s \sim_{\mathcal{M}} s'$), if for every execution sequence $\tau_1^r = s \xrightarrow{o_1}_r s_1 \ldots \xrightarrow{o_n}_r s_n$ there exists a corresponding sequence $\tau_2^r = s' \xrightarrow{o'_1}_r s'_1 \ldots \xrightarrow{o'_n}_r s'_n$ such that $traces(\tau_1^r) = traces(\tau_2^r)$, then for every execution $\tau_1^t = s \xrightarrow{o_1}_t s_1 \ldots \xrightarrow{o_n}_t s_n$, there must also exist a corresponding $\tau_2^t = s' \xrightarrow{o'_1}_t s'_1 \ldots \xrightarrow{o'_n}_t s'_n$ such that $traces(\tau_1^t) = traces(\tau_2^t)$.*

A common strategy to prevent cache timing side channels in the literature is the *constant time* (CT) policy [15], which requires that memory accesses and control flow decisions should depend only on public (non-secret) information. In this paper, the observational model $\mathcal{M}_{ct}$ formalizes this policy and it makes the program counter of each instruction and the accessed memory addresses observable.

Alas, speculative execution introduces new attack vectors that break the assumptions of CT execution. *Spectre* attacks [57] exploit speculation to leak data through side channels like caches. These attacks are characterized by a speculative primitive that allows leaking secrets during speculative execution. We use the observational model $\mathcal{M}_{spec}$ proposed by Buiras et al. [30] to capture *Spectre-PHT/Spectre-V1-style* leakage, i.e., leakage caused by transient execution along a mispredicted conditional branch. The model introduces *refined* (a.k.a., *shadow*) observations that represent operations executed transiently on the misspeculated path.

Observation refinement. Technically, an observation model $\mathcal{M}$ groups states into equivalence classes where states appear indistinguishable. Observation refinement improves this partitioning by introducing a refined model $\mathcal{M}'$ that further partitions these classes. Essentially, $\mathcal{M}'$ captures additional behavioral variations, particularly those linked to side-channel effects, that $\mathcal{M}$ may overlook. For instance, Figure 3 depicts the Spectre V1 primitive annotated with the attacker's observation from the $\mathcal{M}_{ct}$ model and shadow observation from the $\mathcal{M}_{spec}$ that enable the attacker to observe an operation that may execute during the speculation. The current implementation of our toolchain focuses on branch misprediction; modeling other Spectre variants (e.g., store-to-load forwarding in Spectre-v4, or return-stack-buffer effects) would require developing suitable observation models.

Pseudo-(ARM) Assembly	Observations
`ldr x2, [x0]`	load from x0 ($\mathcal{M}_{ct}$ observation)
`if x0 < x1`	branch on x0<x1 ($\mathcal{M}_{ct}$ observation)
`  ldr x3, [x2]`	load from x2 ($\mathcal{M}_{ct}$ observation)
`else`	
`  x2★ = x2`	none
`  x3★ = x3`	none
`  ldr x3★, [x2★]`	load from x2★ ($\mathcal{M}_{spec}$ observation)

Figure 3: The Spectre V1 example instrumented via $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$. We marked shadow observations with ⋆.

3 Methodology

To reason about crypto protocols' resilience to hardware side-channels, we combine symbolic execution, observational models, and protocol verification techniques—the source code of our framework is available at [75]. As Figure 4 illustrates, our approach begins with reverse-engineering the executable binaries using Ghidra [44], after which we transpile the resulting assembly code into **BIR** using the HolBA framework [62]. We then annotate the **BIR** program with attacker observations and symbolically execute it to explore all execution paths. We simplify and translate the resulting symbolic execution tree into a formal model in SAPIC$^+$ calculus, which we then further simplify to improve reasoning scalability. Finally, we translate this model into formats compatible with automatic analysis of side-channel resilience using DEEPSEC.

In the following, we first describe how we annotate **BIR** with observational models to capture side-channel leaks. Next, we detail our symbolic execution engine, tailored for dealing with crypto primitives and hardware interactions. We then explain the extraction and simplification of the protocols model. We conclude by discussing how we leverage DEEPSEC to analyze the simplified SAPIC$^+$ models for side-channel resilience.

Running example. We use the Basic Access Control (BAC) protocol as a running example. Figure 5 shows a C implementation of the BAC's session-establishment logic—developed in-house since production e-passport code is not publicly available—along with the simplified assembly fragment used throughout this section. Our implementation follows the BAC protocol and complies with the relevant International Civil Aviation Organization standards specifications [48]. BAC is a three-pass challenge-response protocol for mutual authentication between an e-passport and a reader. The reader first sends a challenge; the e-passport responds with a fresh nonce. The reader then samples its own nonce, encrypts both nonces under the pre-shared encryption key k_e, and transmits the ciphertext together with a MAC computed under k_m. Upon receipt, the e-passport verifies the MAC, decrypts the ciphertext, and checks that its nonce is present. If all checks succeed, the reader is authenticated (and the e-passport is authenticated to the

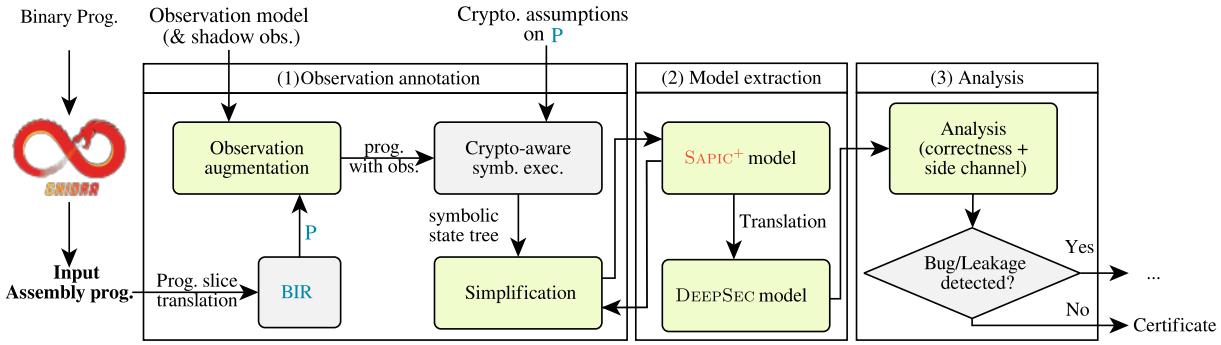

Figure 4: Organization of our approach; new features are in green. We reuse CRYPTOBAP's model-extraction core, but add observation instrumentation, simplification rules, and a DEEPSEC backend for leakage detection.

C Code

```
int passport(char k_e, char k_m) {
 received = get_challenge();
 if (compare(received,"CHALLENGE") == 0){
  gen_n_t = get_nonce();
  send_nonce(gen_n_t);
  cipher = get_msg();
  mac_input = get_msg();
  calc_mac = mac(cipher, k_m);
  if (compare(calc_mac, mac_input) == 0){
   if (sdec(cipher,k_e,xn_r,xn_t,xk_r)){
    if (compare(xn_t, gen_n_t) == 0) {
     k_t = get_nonce();
     response=senc(gen_n_t,xn_r,k_t,k_e);
     mac_resp = mac(response, k_m);
     send(response, mac_resp);
    } else return 0;
   } else return 0;
 } else return 0;}}
```

Simplified Assembly

```
.
.
bl  0x180 <get_nonce>
strb w0, [sp, #16]
add x0, sp, 0x128
bl  0x20 <send_nonce>
bl  0x52 <get_msg>
str x1, x0
bl  0x52 <get_msg>
bl  0x160 <mac>
bl  0x120 <compare>
cmp x0, 0x0
bne  0x200 <exit>
ldr x2, [sp]
.
.
```

Figure 5: Running example. The assembly snippet corresponds to the highlighted C code.

reader analogously). In Figure 6, we highlight the corresponding BIR blocks to illustrate the steps of our methodology.

3.1 Reverse Engineering

Analyzing real-world, closed-source binaries requires first isolating the protocol logic inside a larger code base. In WhatsApp Desktop, the code that implements the Signal/Sesame protocol stack constitutes only a small fraction of the overall executable, yet it is intertwined with application logic, storage, and networking.

Binary extraction. We use Ghidra [44] reverse-engineering platform to (i) locate protocol-relevant entry points, (ii) follow cross-references and call graphs to collect dependent functions, and (iii) export the *exact instructions* of the selected region. We start from functions that (a) retain symbols or strings associated with the Signal protocol stack and (b) interact with crypto-library entry points and network I/O, which can be recognized using approaches in the spirit of [70]. We then expand this set by following data and control dependencies (slicing) to obtain a region that covers message parsing/serialization, state updates, and calls to cryptographic primitives.

Correctness considerations. We treat the reverse-engineering step as part of the trusted computing base: if the selected region omits relevant dependencies, the extracted model may be incomplete. To mitigate this, whenever we cannot resolve a dependency (e.g., an external library call), we conservatively overapproximate it by modeling its outputs as fresh symbolic inputs from the environment. Importantly, we do *not* rely on Ghidra's decompiler for semantic reasoning, i.e., after exporting instructions, all semantic lifting and symbolic reasoning are performed on HolBA's verified BIR semantics. Finally, speculation is accounted for *after* lifting by instrumenting BIR with shadow observations. Thus, we do not require the decompiler to predict speculative paths.

3.2 BIR with Observation Models

To integrate side-channel leakage into our analysis, we instrument BIR programs by annotating each BIR block with attacker observations, which are expressions that describe information that may leak through side channels. These observations can be conditional, meaning they only occur if specific conditions hold in the current state. We adopt established observational models from the Scam-V platform [76]. Our *observation set* O^b includes:

$$O^b = \varepsilon \mid \mathbf{Ld}(\mathrm{a}) \mid \mathbf{St}(\mathrm{a}) \mid \mathbf{Cnd}(\phi, \mathrm{a}_1, \mathrm{a}_2) \mid \mathbf{Pc}(\mathrm{a})$$

Observation Pc(a) exposes the label of the BIR block, which corresponds to the program counter, Cnd(ϕ, a_1, a_2) reveals the outcome of the conditional branch's condition and exposes the addresses of each branch, and Ld(a)/St(a) exposes the address operand of load/store instructions. All other instructions are considered non-leaking and emit the empty observation ε.

Depending on the observation models ($\mathcal{M}_{ct}$, $\mathcal{M}_{spec}$, or any other), the observations assigned to each BIR block can be either normal or shadow observations (see Section 2.3). For instance, take the last line in Figure 3, the $\mathcal{M}_{spec}$ observation annotated to the BIR block of the corresponding assembly code is Ld(x2*).

Compared to other platforms, Scam-V implements a more detailed observation of load and store statements. While we simplify the presentation, we preserve the same level of granularity.

	BIR Block	BIR Observation	SBIR Observation	Sapic$^+$ Process (and similarly, DeepSec Process)
0	`[R30=1;](0x90,jmp(0x52)) // get-msg`	Pc(0x90)	In($R0_1$)	out(0x90); in($R0_1$); //mac of cipher
1	`(0x94,assign(R1, R0))`	Pc(0x94)	Asn($R1, R0_1$)	out(0x94); let $R1 = R0_1$ in
2	`[R30=3;](0x98,jmp(0x52)) // get-msg`	Pc(0x98)	In($R0_2$)	out(0x98); in($R0_2$); //cipher
3	`[R30=4;](0x9c,jmp(0x160))// mac`	Pc(0x9c)	FCall(mac, $R0_2, R0_3$)	out(0x9c); let $R0_3$ = mac($R0_2$) in
4	`[R30=5;](0xa0,jmp(0x120))// compare`	Pc(0xa0)	FCall(compare, $R0_3, R1, R0_4$)	out(0xa0); let $R0_4$ = compare($R0_3, R1$) in
5	`(0xa4,assign(Z, (R0 Equal 0x0)))`	Pc(0xa4)	Asn(Z, ($R0_4$ Equal 0x0))	out(0xa4); let Z = equal($R0_4$, 0x0) in
6	`(0xa8,cjmp(Z, 0xac, 0x200))`	Pc(0xa8), Cnd(Z, 0xac, 0x200)	Cnd(Z, 0xac, 0x200)	out(0xa8); out(Z); if Z
7	`(0xac,assign(R2,load(mem,SP,64)))`	Pc(0xac), Ld(SP)	Asn($R2$,load(mem, SP, 64))	then out(0xac);out(SP);let $R2$ = load(mem, SP) in
	...	...	...	... else out(0x200); 0.

Figure 6: Excerpt of the BIR blocks of the BAC protocol in Figure 5, together with the corresponding BIR observations, symbolic execution events and extracted Sapic$^+$ model. $0x200$ is the address of the BIR halt statement, which translates to 0. The Sapic$^+$ process then translates to the DeepSec process, which closely mirrors what is illustrated here. Jumps (at lines 0, 2, 3, and 4) are the translation of *branch and link* instruction used for function calls in ARM, which requires updating the *link register* $R30$. We present this register update in [..] to mean that it is not relevant to what we intend to present in this example.

The BIR transition relation $\xrightarrow{o^b} \subseteq S^b \times O^b \times S^b$ defines how states evolve while releasing observations. Here, S^b is a set of BIR states, and O^b denotes the set of observations. Starting from an initial state $s_0^b \in S^b$, a sequence of transitions produces a BIR observation trace $t^b = o_1^b, \ldots, o_m^b$. Figure 6 illustrates a BIR program snippet (first column) and its annotated observations (second column).

3.3 Symbolic Execution

To analyze side-channel leakages across all feasible paths, we extend CryptoBap's symbolic execution to track attacker observations alongside path constraints. Each symbolic state $s^s \in S^s$ now encapsulates a path condition—a logical constraint indicating a condition under which a path is taken—and a list of symbolic observations which represent leaked information (e.g., memory addresses and branch targets) that an attacker could infer.

Let $\xrightarrow{o^s} \subseteq S^s \times O^s \times S^s$ denote the transition relation of SBIR and let an SBIR trace be a sequence of observations such that $t^s = o_1^s, \ldots, o_m^s$. SBIR observations $o^s \in O^s$ include BIR observations on symbolic values or expressions, as well as observations related to network communication and calls to crypto primitives, event functions and random number generation. In(x) represents the incoming message x from the environment, whereas Out(x) denotes a message x dispatched to the environment. Fr(n) indicates when the program generates a random number n, and FCall(f, $x_1, \ldots, x_m, y$) refers to a function call f with inputs $x_1, \ldots, x_m$, and an output y. In addition, Ev(e) signifies the occurrence of a visible event e, Loop denotes the start of a loop, and Asn(x, e) represents the assignment of the BIR expression e to the variable x. The SBIR observations of our example are depicted in Figure 6 (third column).

3.4 Model Extraction

To analyze protocol implementations for side-channel leakages, we symbolically execute the instrumented program and derive a symbolic execution tree T from the SBIR execution, with root denoting the initial symbolic state. We use T to extract the protocol's Sapic$^+$ model. The tree captures all feasible execution paths explored during symbolic analysis, annotated with attacker observations and protocol-specific events.

The execution tree T consists of (a) leaves (Leaf), indicating the end of a complete execution path in the tree, i.e., where the halt statement is encountered, (b) event nodes (node(ev) :: T′) each containing a sub-tree T′ and an event ev, and (c) branch nodes (Branch(Cnd(ϕ, a_1, a_2), T_1, T_2)) each with the condition ϕ and subtrees T_i with their respective addresses a_i.

Our symbolic execution generates two successor states for the nodes representing a branching statement (i.e., cjmp), and we continue constructing subtrees from these states. For other statements' nodes, we get an event node with either one or no successor in the tree. The exceptions are the nodes containing *indirect jumps* which may have multiple successors that are discovered iteratively using an SMT solver following the approach outlined in [73].

Having constructed T, we extract the protocol model by translating T into its Sapic$^+$ model using the rules in Figure 7. The leaves in the tree are translated into a null process 0, and events ev in the event nodes are translated into their corresponding Sapic$^+$ constructs. For example, we translate the attacker observations Ld(a), St(a), and Pc(a) within event nodes to the out(⟨a⟩) constructs. The observations Cnd(ϕ, a_1, a_2) are preserved in the branching nodes of T, where each condition ϕ is a symbolic BIR expression translated into an equivalent Sapic$^+$ term. To maintain the attacker's observations during simplification (cf. Section 3.5), we first disclose the translation of ϕ to the attacker before translating the branching node into the Sapic$^+$ conditional construct, where ⟨ϕ⟩ serves as its condition. Each branch of this construct includes the translation of the corresponding addresses a_i—also revealed to the attacker—and the translation of respective subtrees T_i. Figure 7 presents the standard rules for translating expressions.

The more interesting case is the translation of function applications used, e.g., to translate memory load/store and bitwise operations. For instance, a load(mem, a, l), where $l \in \{1, 8, 16, 32, 64, 128\}$, translates to load($mem, a$), with mem and a representing symbolic values for the memory and the address respectively.

The fourth column in Figure 6 shows the extracted Sapic$^+$ process of our running example. An attacker capable of measuring microarchitectural states can detect if the MAC check fails (by observing program counter $0x200$) or succeeds (by observing program counter $0xac$). However, this is not visible without accounting for

$\mathbf{T} = \mathbf{Leaf} \mid \mathbf{node}(\mathbf{ev}) :: \mathbf{T}' \mid \mathbf{Branch}(\mathbf{Cnd}(\phi, \mathbf{a}_1, \mathbf{a}_2), \mathbf{T}_1, \mathbf{T}_2)$ event tree

$(\!|\mathbf{Leaf}|\!) \mapsto 0$

$(\!|\mathbf{node}(\mathbf{ev}) :: \mathbf{T}'|\!)$:= events nodes

$(\!|\mathbf{node}(\varepsilon) :: \mathbf{T}'|\!)$	$\mapsto$	$(\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{o}^s) :: \mathbf{T}'|\!), \mathbf{o}^s \in \{\mathbf{Ld}(\mathbf{a}), \mathbf{St}(\mathbf{a}), \mathbf{Pc}(\mathbf{a})\}$	$\mapsto$	$\mathsf{out}((\!|\mathbf{a}|\!)); (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{Ev}(e)) :: \mathbf{T}'|\!)$	$\mapsto$	$\mathsf{event}\ e; (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{In}(x)) :: \mathbf{T}'|\!)$	$\mapsto$	$\mathsf{in}(x); (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{Out}(x)) :: \mathbf{T}'|\!)$	$\mapsto$	$\mathsf{out}(x); (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{Asn}(x, \mathbf{e})) :: \mathbf{T}'|\!)$	$\mapsto$	$\mathsf{let}\ x = (\!|\mathbf{e}|\!)\ \mathsf{in}\ (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{Fr}(n)) :: \mathbf{T}'|\!)$	$\mapsto$	$\mathsf{new}\ n; (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{Loop}) :: \mathbf{T}'|\!)$	$\mapsto$	$!\ (\!|\mathbf{T}'|\!)$
$(\!|\mathbf{node}(\mathbf{FCall}(\mathrm{f}, x_1, \ldots, x_n, y)) :: \mathbf{T}'|\!)$	$\mapsto$	$\mathsf{let}\ y = \mathrm{f}(x_1, \ldots, x_n)\ \mathsf{in}\ (\!|\mathbf{T}'|\!)\ \mathsf{else}\ 0$

$(\!|\mathbf{Branch}(\mathbf{Cnd}(\phi, \mathbf{a}_1, \mathbf{a}_2), \mathbf{T}_1, \mathbf{T}_2)|\!) \mapsto$
$\mathsf{out}((\!|\phi|\!)); \mathsf{if}\ (\!|\phi|\!)\ \mathsf{then}\ \mathsf{out}((\!|\mathbf{a}_1|\!)); (\!|\mathbf{T}_1|\!)\ \mathsf{else}\ \mathsf{out}((\!|\mathbf{a}_2|\!)); (\!|\mathbf{T}_2|\!)$

$(\!|\phi \in \mathbf{Bexp}|\!)$:= Expressions

$(\!|\mathbf{b} \in \mathbf{Bval}|\!)$	$\mapsto$	$'b' \in \mathcal{N}_{\mathsf{pub}}$	
$(\!|\mathbf{var}\ x|\!)$	$\mapsto$	$x \in \mathcal{V}$	
$(\!|\phi_1 \diamond_b \phi_2|\!)$	$\mapsto$	$(\!|\diamond_b|\!)((\!|\phi_1|\!), (\!|\phi_2|\!))$	Binary operations
$(\!|\diamond_b|\!)$	$\mapsto$	equal: **Equal**; plus, mult, ...: **Plus**, **Mult**, ...	
$(\!|\diamond_u \phi'|\!)$	$\mapsto$	$(\!|\diamond_u|\!)((\!|\phi'|\!))$	Unary operations
$(\!|\diamond_u|\!)$	$\mapsto$	not: **Not**; $\bot$: otherwise	
$(\!|\mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_m)|\!)$	$\mapsto$	$(\!|\mathbf{f}|\!)((\!|\mathbf{e}_1|\!), \ldots, (\!|\mathbf{e}_m|\!))$	

Figure 7: Translating T into a SAPIC$^+$ model: $e, x, x_1, \ldots, x_n, y \in \mathcal{V}$ are variables, $n \in \mathcal{N}_{\mathsf{priv}}$ is a secret name, and not, equal, plus, mult, $\mathrm{f} \in \mathcal{F}$ are function symbols. $\mathbf{o}^s$ and conditional observations are either normal or shadow observations.

side-channel observations, as the program halts when the MAC check fails.

3.5 Simplification

The extracted models include memory operations and attacker-observable events, and are therefore too large for current protocol provers. To reduce model complexity, we used abstractions in BIR, pruned paths at SBIR, and applied several simplification rules at the SAPIC$^+$ level, which are shown in Table 1.

At the BIR level, we introduce a *storage abstraction* similar to how KLEE abstracts files, pipes, and terminals [31]. We abstract the SQLite database engine by explicitly modelling the effects of database creation, the table schema, and the read and write operations on the database file. In particular, WhatsApp stores session keys in the SQLite database.

HolBA's semantic-preserving transpiler inserts assertions into BIR programs that encode *well-formedness invariants* of executions, e.g., after each stack operation it adds $\mathbf{assert}(sp_{low} \leq sp \leq sp_{high})$ to confine the stack pointer to the current frame. By construction, each inserted $\mathbf{assert}(\chi)$ is an invariant for executions of the original binary that start from well-formed initial states. Consequently, any SBIR path that violates such an assertion is infeasible. We therefore cause the path to fail at that point and prune its suffix. This does not affect our side-channel analysis because the inserted assertions hold for all executions from well-formed states, and our observation

SAPIC$^+$ Process	if ϕ then P else P	let ϕ in 0 else 0	let $x = e$ in P s.t. $x \notin vars(\mathrm{P})$
Simplified	P	0	P

Table 1: Simplification rules: *vars* are a set of variables for a given process.

semantics are guarded. Hence, assertion-violating paths cannot produce observations.

At the SBIR level, further simplifications are possible. An example is pruning paths that the SMT solver marks as unreachable, like branches of conditionals that are always false. While for functional correctness analysis, eliminating such paths is possible, we cannot apply this simplification when we look for side-channel leaks, as unreachable branches can leak during speculative execution.

At the SAPIC$^+$ level, nil processes indicate the end of a complete path. When a let-binding 'let $x = e$ in P else Q' results in nil processes in both its success and failure branch (i.e., $\mathrm{P} = 0$ and $\mathrm{Q} = 0$), it can be removed and substituted with 0. Moreover, for a conditional that involves paths following the same sequence of constructs, like if ϕ then P else P, the conditional can be replaced with the corresponding constructs in those paths, P. In order to ensure that this simplification would not affect our side channel analysis, we preserve the observations related to the conditional (if any) and add them to the simplification result.

Live variable analysis is a data-flow analysis used in compiler optimization to identify live variables at every point in a program [8]. A variable x is live at a given point p in the program if x will be used along some path starting from p. This analysis becomes especially important when we want to eliminate a let construct. We use this technique to determine whether the variable x used in the let $x = e$ in P construct remains alive within the sub-process P. If so, we retain the let construct, otherwise, we substitute it to P.

Finally, observational models like $\mathcal{M}_{ct}$ require making the program counter of instructions (i.e., $\mathbf{Pc}(\mathbf{a})$) visible to the attacker. When extracting a formal model of a program annotated with such observational models, we generate an $\mathsf{out}((\!|\mathbf{a}|\!))$ for each instruction (see Figure 7). However, the large number of output constructs makes DEEPSEC computation expensive without a significant benefit, mainly when they originate from the same branch. Therefore, leaking the program counters of conditional branches is sufficient and other program counter leakages can be eliminated. This greatly simplifies the extracted SAPIC$^+$ process (e.g., see Table 3).

3.6 Detecting Leakages with DEEPSEC

We use the protocol verifier DEEPSEC [37] to detect side-channel leakage exploitable by a Dolev–Yao network attacker who can additionally observe the leakage events produced by our contracts. DEEPSEC specializes in privacy properties expressed as trace equivalence: given two processes P_1 and P_2, it checks whether an attacker can distinguish their observable execution traces.

For instance, in BAC, an attacker can exploit observable control-flow differences (e.g., $\mathsf{out}(0xac)$ versus $\mathsf{out}(0x200)$ in Figure 6) to infer information about message structure. By replaying previously observed messages, the attacker can distinguish successful from

unsuccessful MAC verification and thereby violate BAC's intended privacy guarantees.

DeepSec is well-suited for such analyses because it supports equivalence checking. Formally, it decides whether $P_1 \approx P_2$ holds for the class of processes it supports; when it terminates, it either proves equivalence or returns a counterexample in the form of distinguishing traces.

To isolate leakage introduced by speculative execution, we evaluate conditional non-interference (Definition 1) by running the same equivalence check under two observation models. Concretely, we check equivalence once under the constant-time observations $\mathcal{M}_{ct}$ and once under the refined speculative observations $\mathcal{M}_{spec}$. If $P_1 \sim_{\mathcal{M}_{ct}} P_2$ but $P_1 \not\sim_{\mathcal{M}_{spec}} P_2$, the additional distinguishing power comes from speculation, witnessing a speculative-only leak.

A counterexample under $\mathcal{M}_{ct}$ (resp. $\mathcal{M}_{spec}$) yields two symbolic traces distinguishable by the chosen observation model; under the assumptions of our lifting and abstractions, this corresponds to a concrete binary-level side-channel distinguisher within our leakage contract. Conversely, when DeepSec reports equivalence, the result is bounded (due to bounded replication) and depends on the soundness of our abstractions for cryptographic primitives, external I/O, and the selected observations.

4 Evaluation

In our evaluation, we separate two goals that require different backends and abstractions: (i) *reachability-style* security analysis of WhatsApp Desktop (e.g., forward secrecy under a Dolev–Yao network attacker), and (ii) *equivalence-style* privacy and side-channel analysis (unlinkability and conditional non-interference) under the observation models $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$. For (i), we reuse CryptoBap's established extraction pipeline and discharge the resulting models to ProVerif and Tamarin. For (ii), we apply our observation-aware extraction (Sec. 3) and analyze the resulting Sapic$^+$ processes with DeepSec.

Table 2 summarizes the sizes and extraction/verification costs for the WhatsApp components (Sesame session handling and double ratchet). Table 3 summarizes the corresponding statistics for the side-channel case studies (BAC and WhatsApp session establishment) under $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$. The reduction percentages report the relative decrease from the raw extracted model (*all*) to the simplified model (*simp*). Note that verification time may increase because simplification and backend translation add overhead.

Manual vs. automatic effort. Once a protocol-relevant region is selected, the pipeline (lifting, observation instrumentation, symbolic execution, extraction, simplification, translation, and verification) runs automatically. The manual inputs are: (a) selecting entry points and defining the analysis boundary in Ghidra, (b) deciding which external calls are treated as cryptographic abstractions or environment stubs, and (c) stating the verification query for the chosen backend. For BAC, this required a few hours of reverse engineering and modeling. For WhatsApp, locating and validating protocol-relevant entry points in Ghidra required a few days; after that, the remainder of the pipeline was fully automated.

4.1 Evaluation of WhatsApp with CryptoBap

WhatsApp allows users to exchange messages, share status posts, and make audio and video calls. Since 2016 [90] WhatsApp uses a modified version of the Open Whisper Systems' Signal protocol as the basis for end-to-end encryption. This encryption protocol prevents WhatsApp's servers and other third parties from accessing the plaintext of user messages or calls. Messages are encrypted with ephemeral cryptographic keys that are regularly updated (using ratcheting or a new handshake), preventing attackers from decrypting previously transmitted messages, even if the current encryption keys are compromised.

We extract a formal model of WhatsApp's binary to verify forward secrecy and post-compromise security. For this part, we use CryptoBap's original extraction pipeline to obtain a Sapic$^+$ model that can be discharged to ProVerif and Tamarin for reachability-style analysis; this analysis is orthogonal to the side-channel observations studied later in this section. We reverse-engineered the WhatsApp Desktop's binary using Ghidra. Unlike the iOS or Android versions, function symbols were not stripped from the desktop application, allowing us to match the function names with the latest version of the `libsignal`, Signal's protocol implementation.

4.1.1 Components. The double ratchet protocol enables secure (confidential and authentic) message exchange between the two parties. It builds on the Extended Triple Diffie-Hellman (X3DH) key agreement protocol [55], which allows parties to establish a shared secret key using mutual authentication based on their public keys.

Initially, the key is obtained via X3DH, and called the *root key*. Subsequent ephemeral keys are generated using ratcheting, so called because earlier ephemerals cannot be derived from later ones. Ratcheting is *asymmetric* if the parties switch roles (i.e., from sender to receiver or vice versa) and *symmetric* if they maintain the same role. Symmetric ratcheting provides *forward secrecy*: even if long-term secrets (the secret keys to the public keys used in the handshake) or future ephemerals are revealed, past ephemerals and thus the messages sent with them remain secret. Asymmetric ratcheting establishes fresh ephemerals that are unknown to the attacker even if *past* ephemerals are known, unless the attacker actively attacks the handshake. This provides *post-compromise* security: even after ephemeral keys are revealed, the authenticity and confidentiality of future ephemerals (and the messages encrypted with them) can be recovered.

Sesame defines the session management and operates on the layer above the double ratchet protocol. It manages multiple double ratchet sessions between the different devices associated with each user account. E.g., if Alex has n_A devices and Blake has n_B devices, on each of Alex's devices, Sesame manages n_B connections to Blake and n_A-1 connections to Alex's other devices. Sesame manages the creation, deletion, and use of sessions, maintaining a local database that records each party's devices and their associated sessions. To establish a pairwise encrypted conversation between two users, Sesame manages an instance of the double ratchet protocol (including X3DH key agreement) between each of their devices.

4.1.2 Extracted Model. We abstract the X3DH key agreement protocol with a private channel that 'magically' communicates a master

WhatsApp Protocol	#ARM	#symb. exec. paths		#Sapic+ LoC			#TM	#PV	time to extract			Overall verif. time	
	LoC	feasible	infeasible	all	simplified	crypto.	LoC	LoC	all	simplified	crypto.	TM	PV
(1) Initiate Session	6844	12	25	106	61 (42% ↓)	33	161	171	13	14 (7% ↑)	12	19.76	0.057
(2) Respond to (1)	5803	106	413	718	92 (87% ↓)	12			65	60 (7% ↓)	38		
(3) Send Message	3041	156	368	983	429 (56% ↓)	166			71	100 (40% ↑)	65		
(4) Receive Message	4181	7293	19322	31257	2033 (93% ↓)	360			26903	40128(49% ↑)	25782		

Table 2: WhatsApp protocol analysis. The table includes, the size of the code under analysis, the explored symbolic paths, the extracted models—complete and simplified, consisting solely of cryptographic operations, and translated into Tamarin and ProVerif—along with the duration (in seconds) of each step. TM and PV refer to Tamarin and ProVerif, respectively.

secret key (initial root key) between two parties. Like in the handwritten Tamarin model for Signal in [41], this avoids verification issues with Tamarin's limited Diffie-Hellman (DH) theory (see [36, Sec. 2.3 and 2.4] for further discussion on the support of DH theories in Tamarin and other protocol verifiers).

Excluding X3DH, we extract a model of the core components:

(1) *initiating a new session* in which the client requests the recipient's keys from the WhatsApp server and calculates a session key,
(2) *responding to a request to initiate a new session* by the recipient and calculating the corresponding session key,
(3) *transmitting messages* in a session (symmetric ratchet), and
(4) *receiving messages* within a session (asymmetric ratchet).

Table 2 presents the data obtained from our evaluation of the WhatsApp components using Tamarin and ProVerif.

By considering all memory operations and function calls (but abstracting cryptographic library calls), we initially obtain a Sapic+ model with nearly 33,000 LoC. Simplification (Section 3.5) reduces this to about 2,600 LoC, but both Tamarin and ProVerif still fail to terminate at this scale. For the reachability analysis reported below, we therefore use CryptoBap's memory-abstraction variant (i.e., without explicit load/store primitives), which yields a substantially smaller Sapic+ model that remains suitable for analyzing security properties under the Dolev–Yao attacker model. We highlighted the memory-aware numbers above to emphasize the scalability bottleneck of state-of-the-art protocol verifiers that currently prevents full memory-precise verification of large real-world binaries.

4.1.3 Implementation vs. Specification. We find that, in certain scenarios, the protocol implementation behaves differently from its specification in the WhatsApp security white-paper [90], in particular the component responsible for decrypting incoming messages (i.e., the component number 4 in Table 2). The whitepaper specifies that the chain and root keys are updated upon receiving a response, yet it lacks a sufficient explanation of the ratcheting mechanism [90, p.15] (in contrast to the Signal documentation [55]). According to WhatsApp, the ratcheting process involves each party calculating its next chain and root keys using a shared ephemeral secret, derived from the sender's and receiver's ephemeral keys, and a root key whose origin is not clearly defined. Instead, our model indicates three distinct behaviors for ratcheting. The next chain key is either derived from: (a) the current chain key, (b) the new chain key, calculated from fresh root and ephemeral keys, and (c) the fresh chain key, calculated from the current root key and a new ephemeral key.

This highlights the strength of model extraction. Most protocol models have to rely on the specification to be correct, even if the source is available, a thorough comparison is tedious and requires tool support (such as model extraction for the source language). Often, behaviors are underspecified (as in this case), and the modeler fills the gap. Sometimes source-code inspection helps fill the gap, but WhatsApp's source code is closed. Ultimately, formal analysis results rely on the adequacy of the model. Gaps through underspecification or mispecification can lead to overlooking attacks.

4.1.4 Authenticity with Tamarin & ProVerif. We used our extracted model to analyze WhatsApp's session management and double ratchet protocol with Tamarin and ProVerif.

Utilizing these tools, we verify the security properties of our extracted model by defining queries and checking the reachability of all events in the model. Our analysis shows that Tamarin and ProVerif successfully verify *forward secrecy* for two parties initiating a session and exchanging secret messages. However, our analysis revealed a major issue that prior work [41] had identified in the Signal application and *suspected* in WhatsApp. *WhatsApp violates post-compromise security despite employing the double ratchet protocol when faced with a clone attacker.* Cremers et al. [41] have proposed secure mechanisms that offer stronger guarantees. However, our analysis showed that WhatsApp does not implement any of the proposals, allowing a clone attacker to break post-compromise security.

Note, while the analysis of Signal's implementation benefits from direct access to the source code (the model in [41] was manually crafted but informed by the code), our models were derived entirely from the binary. Moreover, while ProVerif automatically identifies the post-compromise security violation for WhatsApp, we employed heuristics to guide Tamarin's proof search and manually selected 303 proof steps from the available options.

4.2 Side-Channel Analysis

The second part of our evaluation demonstrates how side-channel analysis of protocol implementations can reveal hardware-induced leakages. Although abstract formal models and implementations of cryptographic protocols are often formally verified against security properties, they have not been analyzed in the presence of hardware-induced leakages. Our methodology addresses this gap by combining model extraction with leakage contracts, enabling automated analysis of side-channel leaks. We focus on BAC and WhatsApp due to their widespread use in real-world scenarios and their critical security implications to evaluate their resilience against potential attacks. In these case studies, we assume the attacker can influence protocol inputs, e.g., by replaying or crafting messages. Additionally, the attacker can observe microarchitectural side effects captured by our observation models, including program counter and load/store addresses, as would be possible for

a co-resident process performing cache attacks. Thus, the detected distinguishers reveal violations that are *triggerable* by network interaction but *observable* through side channels on the victim device.

4.2.1 Basic Access Control Protocol. Many countries are adopting electronic biometric passports, a.k.a. e-passports. These passports encode the holder's digital information within a Radio Frequency Identification (RFID) chip for interaction with passport readers. However, e-passports face several threats related to security and privacy, including skimming, cloning, and eavesdropping [13, 54]. To address these concerns, the International Civil Aviation Organization (ICAO) standardized security mechanisms, among those, the BAC protocol. BAC's goal is to ensure that only authorized parties can access personal information stored in passports' RFID. A party (the reader) is authorized if they know a machine-readable code printed on each e-passport, which the passport holder typically provides by physically handing the passport to the reader or scanning it themselves (if the reader is a device).

Despite the ICAO standard's goal to create an unlinkable BAC protocol, some implementations of the protocol permit reidentification of a passport holder. Arapinis et al. [11] found that these implementations output different error messages if a replayed message is invalid due to an incorrect MAC or an incorrect nonce. As the second case can only occur when the message is replayed to the same passport (because each interaction uses a fresh nonce), while the first can only occur if the message is replayed to a different passport (as the MAC key is fixed per passport), this allows for identification of the passport holder. Subsequently, the error messages were unified, but Chothia and Smirnov [38] find that all e-passports are still vulnerable to a variant of the attack where instead of the error message, the attacker measures the computation time of the passport's response to distinguish the incorrect-MAC branch and the incorrect-nonce branch.

To show how our methodology finds such attacks on critical systems, we implemented the BAC protocol ourselves. While the machine code is not easily extractable from RFID chips for researchers (and the legal implications are unclear), standardization bodies could request access and run similar analyses. As depicted in Figure 5, our implementation does not allow distinction by error message, but rather by side-channel leakage. Using our methodology, we found that an attacker can use this leak to distinguish between both types of failures and thus break unlinkability. The implementation is vulnerable under both $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$, see Table 3.

4.2.2 Privacy Properties with DeepSec. In order to analyze the WhatsApp application vulnerability to hardware-induced leaks, we start by annotating the **BIR** translation of the session initialization binary (component 1) using both the constant-time $\mathcal{M}_{ct}$ and the speculation $\mathcal{M}_{spec}$ models. For each of the two resulting annotated **BIR** programs, we perform symbolic execution, extract its Sapic$^+$ model and translate into DeepSec. The evaluation results for session initialization (component 1) with DeepSec are listed in Table 3.

DeepSec checks trace equivalence (and other process equivalences), which is typically used to encode privacy properties like unlinkability, voting privacy, and more [37]. One of the relevant privacy properties of WhatsApp is *unlinkability*, which states that an attacker cannot distinguish between two sessions—one involving protocol parties Alice and Bob, and another involving Alice and Charlie. In our bounded setting, this standard unlinkability holds for both extracted models (under $\mathcal{M}_{ct}$ and $\mathcal{M}_{spec}$). We additionally consider a refined, application-level privacy property for session establishment: an attacker should not be able to tell whether a user is initiating a conversation with a recipient for the first time (first-contact) or establishing a new session for an existing conversation. DeepSec finds a distinguisher for this refined property already under $\mathcal{M}_{ct}$ (and hence also under $\mathcal{M}_{spec}$), because the implementation's control flow depends on whether a one-time pre-key (OTPK) is present in the recipient's pre-key bundle; such control-flow differences are observable via instruction-cache attacks.

Note that we assume the WhatsApp server is honest in our models and model the communication between WhatsApp clients and the server—which, in reality, is secured using the Noise protocol framework [90, p. 35]—as private channels.

4.2.3 Discovery of Privacy Attack. Our core discovery is a privacy violation in the WhatsApp Desktop application, where an attacker can determine whether a victim (i.e., a WhatsApp account holder) is initiating a conversation with a third party for the first time. When a victim attempts to establish a pairwise encrypted session with the recipient, they retrieve the recipient's *pre-key bundle* from the WhatsApp server. The pre-key bundle includes the public identity key, the public signed pre-key, and a single public one-time pre-key (OTPK) of the recipient [90]. According to the WhatsApp security whitepaper [90], the recipient's public OTPK is used only once and is removed from the server after it has been requested to initiate the first-time conversation. Therefore, whether or not the recipient's public OTPK is present when the victim establishes a new pairwise encrypted session with the recipient distinguishes a first-time conversation from a new session for an existing conversation.

This attack applies to one-to-one conversations as well as to group chats, which makes it more severe. Using our attack, an attacker can create a group with the victim and add any chosen third party, then check if the victim and the selected third party have been in contact. By repeating this process and inviting more third parties to the group chat, the attacker can reconstruct the victim's social graph—that is, the set of individuals with whom the victim communicates. When an attacker reconstructs the victim's social graph, sensitive details about the victim's personal life, including their relationships, interests, and activities, are revealed.

We analyzed the **BIR** program for the WhatsApp component's assembly code, which initiates a secure session by retrieving and processing the recipient's pre-key bundle. By annotating this **BIR** program with observational models and extracting the respective models, we were able to analyze these models using DeepSec (as detailed in Table 3). Our analysis revealed a condition tied to the recipient's public OTPK. As a result, a co-resident attacker process on the victim's system can determine if the recipient's public OTPK exists by observing WhatsApp's control flow.

4.2.4 Attack Vector (Instruction Cache). In our model, the above attack requires the side channel to reveal the program's control flow to learn the presence or absence of the recipient's public one-time pre-key. Both observation models represent realistic attack vectors [30]; we will now outline how our attack can be realized as an *instruction cache attack*. Such attacks exploit the timing differences in the processor instruction cache behavior to infer which program

Protocols	# ARM LoC	# symb. exec. paths		# SAPIC+ LoC		# DEEPSEC LoC	time to extract		Verif. time in DEEPSEC
		feasible	infeasible	all	simplified		all	simplified	
BAC with $\mathcal{M}_{ct}$	176	5	69	242	83 (65% ↓)	205	9	7 (22% ↓)	1
BAC with $\mathcal{M}_{spec}$	194	15	170	630	240 (61% ↓)	589	22	18 (18% ↓)	7
Initiate Session with $\mathcal{M}_{ct}$	6844	12	65	283	90 (68% ↓)	220	43	44 (2% ↑)	2
Initiate Session with $\mathcal{M}_{spec}$	6899	120	679	2872	903 (68% ↓)	2191	148	130 (12% ↓)	5

Table 3: Case studies with observation models. Columns represent: the analyzed ARM code size, the number of analyzed symbolic paths, the extracted models size before and after simplification and after translation to DEEPSEC, and the time (in seconds) required for each step.

paths are executed without requiring direct access to the program counter or memory contents.

WhatsApp Desktop's assembly code includes a conditional branch that checks for the existence of the one-time pre-key in the pre-key bundle during session establishment. When the one-time pre-key is present (indicating a first-time conversation), a specific set of instructions is executed to process it, whereas a different program path is taken if the key is absent (indicating an existing conversation). These two distinct execution paths result in different patterns of instruction cache usage. In our proof-of-concept attack implementation, we used the Prime+Probe technique [82] to monitor the instruction cache by co-locating a process on the same CPU core as the WhatsApp Desktop application. Interested readers can find the details of our PoC in Appendix A.

In a Prime+Probe attack, the attacker first 'primes' the instruction cache by executing their own code to fill specific cache sets. Next, the attacker triggers the victim to establish a new encrypted session, e.g., by initiating a group chat with the victim and inviting the third person. Then, the attacker 'probes' the cache by measuring the time it takes to re-execute their code. If the victim's execution of the pre-key processing code evicts the attacker's instructions from the cache (due to cache set conflicts), the probe will take longer, revealing that the 'first-time conversation' path was taken. Conversely, if the cache remains largely untouched, it indicates the 'existing conversation' path was executed. By mapping the WhatsApp application's instruction addresses to cache sets, the attacker can reliably distinguish between these two cases. The instruction cache attack eliminates the need to observe the program counter in the clear or have privileged access to the victim's system memory.

Real-World Impact. The vulnerabilities we discovered have a real-life impact. Imagine that V is a journalist connected with numerous sources, including whistle-blower T. The attacker R is a regime that enforces its citizens to install a certain application which allows it to mount instruction cache attacks.[2] The regime infiltrates some of V's WhatsApp groups unrelated to their journalistic work. R convinces a moderator in this group to include T in a chat discussion. Using our attack, R confirms that V was in contact with T.

[2]This is plausible, e.g., Russia [19], Kazakhstan [1], Saudi Arabia [5] and Singapore [2] provide certain services like visa or passport requests, tax declaration, or social services exclusively via government-provided Smartphone Apps. In other countries, like Austria, Estonia, or Germany digitalized services with a strong App focus increase the pressure to install such Apps, often in connection to digital ID [18]. Some states in the U.S. also increase the pressure to install Apps for social services [79]. Many more countries had indirect enforcement of the use of Apps during the COVID-19 pandemic, e.g., to access public buildings or for citizens who tested positive [4].

5 Related Work

5.1 Protocol Verification

The usage of formal methods to analyze protocols and verify properties like secrecy and authentication dates back to Lowe's work [65]. Traditionally, domain experts translate protocol specifications into formal models to analyze high-level behavior using theorem proving [17, 81], model checking [16, 65], and symbolic analysis [25, 36, 69]. Tools like PROVERIF [25] and TAMARIN [69] are effective in verifying security properties. But, these specification-based models abstract away implementation details, potentially overlooking vulnerabilities introduced during execution, like hardware-induced leakages [58, 63] or compiler bugs [85, 86].

To address discrepancies between specifications and implementations, techniques like fuzz testing [10, 28], differential testing [66], symbolic execution [9], code generation [6, 32], deductive verification [12, 32, 60], and type checking [23, 24] have been employed, which target the implementation of protocols. Model extraction techniques further aim to verify implementations in high-level languages like *F#* [14, 21, 22], Java [56, 80], C [9, 35, 50], and Rust [60]. Yet, the complexity of these languages limits their practicality.

CRYPTOBAP addresses these gaps by extracting formal models directly from protocol machine code, avoiding reliance on specifications or high-level code. Our work builds on CRYPTOBAP, but extends the analysis along three axes: we (i) begin from the binary of real (closed-source) applications—not the one we compile ourselves, (ii) instrument the lifted binary with leakage contracts (e.g., constant-time and speculative observation models) and extract observation-aware models, and (iii) connect the extracted models to equivalence-based verification in order to reason about protocol properties (e.g., unlinkability) in the presence of microarchitectural observations.

This enables us to use the extracted models to assess (a) whether protocol properties are preserved in the presence of hardware leaks and (b) the existence of microarchitectural leakages through the protocol verifier DEEPSEC. Unlike CRYPTOBAP, we extract the memory operation models into SAPIC+ and apply simplification rules to scale the extracted models for analysis with protocol verifiers.

5.2 Side-Channel Analysis of Crypto Libraries

In recent years, side-channel detection, mitigation, and formal analysis have become active areas of research, with numerous papers published on the topic. Here, we do not intend to survey all works in this domain (an incomplete list can be found [33, 64]); rather, we focus on a few selected works that are closer to our approach.

Crypto libraries have been the prime target of side-channel attacks, e.g., timing attacks. The constant-time (CT) paradigm mitigates these by ensuring control flow and memory accesses are independent of secret data under sequential execution [20]. High-assurance frameworks like Jasmin [78] and FaCT [83] use formal methods, such as information-flow type systems, to enforce CT and produce verified, efficient implementations.

However, Spectre attacks [57], exploiting speculative execution, challenge the CT paradigm by leaking secrets even in CT-compliant code. Research has extended CT to speculative constant-time (SCT) to protect against Spectre variants. Shivakumar et al. [84] introduced a type system in Jasmin for SCT against Spectre-v1 with minimal overhead using selective speculative load hardening. They have also addressed declassification issues under speculation, proposing relative non-interference (RNI) and efficient countermeasures. Arranz Olmos et al. [78] extended this to all Spectre variants, including Spectre-RSB, with low overheads. Other approaches like Swivel [72] and Serberus [71] offer protections but may rely on hardware or incur higher overheads.

SCT-style work primarily targets CT guarantees for *crypto primitive implementations* and often reasons about a concrete source language or assembly semantics. Our work is complementary. We take observation models inspired by Scam-V/SCT-style semantics and lift them to Sapic$^+$ protocol models in order to analyze *protocol-level* properties that can be violated by application (e.g., secret-dependent error handling or first-contact branches), even when cryptographic primitives are treated as ideal. Throughout, we use "hardware-induced" leakage in the microarchitectural sense (e.g., cache-based observations), and do not model physical channels such as electromagnetic (EM) or power.

However, previous studies did not connect microarchitectural side-channel reasoning to protocol-level verification of real protocol *implementations* (as opposed to specifications or isolated primitives); this is the gap we target. In this work, we propose a methodology designed to identify such leakages and address this gap.

6 Discussions

Soundness of Proposed Methodology. Regarding our methodology soundness, transpilation from assembly to **BIR** and symbolic execution remain sound, although we lack formal proofs validating our translation to Sapic$^+$ (see Section 3.4). Since our primary objective with the proposed methodology is attack detection, this is acceptable, as attacks can be manually validated if false positives remain manageable. However, a positive security result from DeepSec cannot be fully trusted. For instance, we obtain a verification result for the strong secrecy of the master key (initial root key) derived during WhatsApp's session initialization with DeepSec, but refrain from reporting it as we cannot be sure it would hold.

Symbolic execution is usually only shown sound, but it is possible to show its completeness w.r.t. assignments consistent with a path condition. If the subsequent translation step can be shown to maintain that pattern, we should be able to show that a symbolic equivalence between the translated processes implies a concrete trace equivalence as long as (symbolic) traces with consistent assignments are only mapped to (symbolic) traces with consistent assignments. In our concrete application, this could be achieved by revealing the path constraints in the trace, as the program is compared to itself and thus the conditionals are observed through the program-counter observations.

Component Selection via Ghidra. We currently identify WhatsApp components manually in Ghidra using (i) preserved function symbols corresponding to the Signal protocol, and (ii) interactions with the crypto library and network I/O, which can be systematically recognized [70]. In principle, information-flow analysis could automate this step by combining existing detection techniques [70] with conservative over-approximation (treating all incoming inputs as arbitrary). We are working toward integrating such automation.

Mitigation Against Side-channel Attacks. Side-channel risks can be reduced either by constant-time rewrites (eliminating secret-dependent branches and memory access patterns) or by removing the cache channel (e.g., isolation/partitioning or cache flushes). Implementations with effective isolation, partitioning, or flushing are not vulnerable under our observation-based model. Our toolchain can validate *code-level* mitigations by re-extracting the binary and re-running DeepSec equivalence under $\mathcal{M}_{ct}$ and then $\mathcal{M}_{spec}$. However, we cannot currently verify fence-based mitigations or hardware/OS defenses; doing so would require extending the underlying semantics and observation models.

Limitations of Protocol Verifiers. The extracted models can reach the practical limits of current protocol verifiers. Even hand-written and optimized models (e.g., for TLS 1.3 [40], WPA2 [42], and SPDM [39]) may require hours to days of verification time, substantial memory, and extensive proof guidance. These tools have improved on *protocol complexity*, but not primarily on scaling with the *size* of automatically generated rule sets. Consequently, our methodology would benefit from better support for decomposition and modular verification, which remains limited in existing tools.

Dependence on Crypto Libraries. Our analysis is conducted in the Dolev–Yao model and therefore assumes crypto libraries are trusted. If primitives deviate due to bugs, unexpected weaknesses, or their own side channels, our conclusions may not hold.

Applicability Constraints. Our approach assumes access to the target binaries and reliance on a known cryptographic library. In some settings (e.g., e-passport implementations), external researchers may not have access to code at all. While WhatsApp employs basic obfuscation, we were still able to extract meaningful models; more targeted anti-analysis obfuscation could impede extraction, though such measures may themselves be suspicious.

ISA and scalability limitations. Our implementation targets AArch64 and RISC-V because HolBA provides lifting for these ISAs. Supporting additional ISAs (e.g., x86-64) requires a lifter with comparable semantic assurance. Scalability is limited by symbolic execution and by the size of extracted Sapic$^+$ models, especially when modeling memory explicitly. This motivates using a memory abstraction for full-program reachability in WhatsApp, and focusing observation-aware side-channel analyses on smaller components.

Scope of side-channel modeling. Our leakage contracts expose a small observation set (program counter information, symbolic load/store addresses, plus speculative "shadow" observations).

This captures many cache-based attacks, but not all physical or microarchitectural effects. Thus, a found distinguisher indicates a concrete leak, whereas a proved equivalence does not exclude channels outside the observation set.

7 Concluding Remarks

We presented a methodology for analyzing crypto protocol *implementations* that combines binary-level model extraction with protocol verification under explicit leakage contracts. Our approach extends CryptoBap with attacker observation instrumentation, a Ghidra-assisted front-end for locating protocol code in real binaries, and an equivalence-checking backend (DeepSec) for privacy and conditional non-interference.

We extracted an implementation-level model of WhatsApp Desktop's session management and double ratchet components and verified forward secrecy, while confirming a post-compromise security break against a clone attacker. For side-channel, we found a new instruction-cache leak in WhatsApp session establishment that enables social-graph inference and confirmed the known unlinkability break in BAC under microarchitectural observations.

Our results suggest that protocol implementations deserve the same rigor of side-channel analysis traditionally applied to cryptographic libraries: protocol verifiers provide principled, composition-aware notions of secrecy and privacy, and lifting these notions to implementation traces makes it possible to detect protocol-level violations that are invisible in specification-only models.

Acknowledgments

We thank anonymous reviewers for their insightful comments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We also gratefully acknowledge a gift from Intel and Amazon.

References

[1] 1992. E-government. https://en.wikipedia.org/wiki/E-government
[2] 2003. Singpass Singapore's National Digital Identity (Factsheet). https://www.mddi.gov.sg/newsroom/singpass-factsheet-02032022
[3] 2009. WhatsApp. https://www.whatsapp.com/about
[4] 2020. TraceTogether. https://en.wikipedia.org/wiki/TraceTogether
[5] 2021. Absher. https://en.wikipedia.org/wiki/Absher_%28application%29
[6] Coşku Acay, Joshua Gancher, Rolph Recto, and Andrew C. Myers. 2024. Secure Synthesis of Distributed Cryptographic Applications. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 433–448. doi:10.1109/CSF61375.2024.00021
[7] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-driven Cache Attacks on AES (Short Paper). In *Proceedings of the 8th International Conference on Information and Communications Security* (Raleigh, NC) *(ICICS)*. Springer-Verlag, 112–121.
[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
[9] Mihail Aizatulin. 2015. *Verifying Cryptographic Security Implementations in C Using Automated Model Extraction*. Ph. D. Dissertation. The Open University. http://arxiv.org/abs/2001.00806
[10] Max Ammann, Lucca Hirschi, and Steve Kremer. 2024. DY fuzzing: formal Dolev-Yao models meet cryptographic protocol fuzz testing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1481–1499.
[11] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. 2010. Analysing unlinkability and anonymity using the applied pi calculus. In *2010 23rd IEEE computer security foundations symposium*. IEEE, 107–121.
[12] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David Basin, and Peter Müller. 2022. Sound Verification of Security Protocols: From Design to Interoperable Implementations (Extended Version). arXiv:2212.04171 [cs]
[13] Gildas Avoine, Antonin Beaujeant, Julio Hernandez-Castro, Louis Demay, and Philippe Teuwen. 2016. A survey of security and privacy issues in ePassport protocols. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–37.
[14] Michael Backes, Matteo Maffei, and Dominique Unruh. 2010. Computationally sound verification of source code. In *Proceedings of the 17th ACM conference on Computer and communications security*. 387–398.
[15] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1267–1279.
[16] David Basin, Cas Cremers, and Catherine Meadows. 2018. Model Checking Security Protocols. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, Cham, 727–762. doi:10.1007/978-3-319-10575-8_22
[17] David Basin, Andreas Lochbihler, Ueli Maurer, and S Reza Sefidgar. 2021. Abstract modeling of system communication in constructive cryptography using CryptHOL. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 1–16.
[18] Marc Bennetts. 2025. E-government in Europe. https://en.wikipedia.org/wiki/E-government_in_Europe
[19] Marc Bennetts. 2025. Putin launches spy app to keep Russians in 'digital gulag'. https://www.thetimes.com/world/russia-ukraine-war/article/putin-moscow-whatsapp-ban-plan-max-app-launch-b789tt6ts
[20] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3958)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer, 207–228. doi:10.1007/11745853_14
[21] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 483–502.
[22] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. 2006. Verified reference implementations of WS-Security protocols. In *International Workshop on Web Services and Formal Methods*. Springer, 88–106.
[23] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. 2010. Modular verification of security protocol code by typing. *ACM Sigplan Notices* 45, 1 (2010), 445–456.
[24] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. 2016. miTLS: Verifying Protocol Implementations against Real-World Attacks. *IEEE Security & Privacy* 14, 6 (2016), 18–25. doi:10.1109/MSP.2016.123
[25] Bruno Blanchet. 2022. The Security Protocol Verifier ProVerif and Its Horn Clause Resolution Algorithm. *Electronic Proceedings in Theoretical Computer Science* 373 (Nov. 2022), 14–22. doi:10.4204/EPTCS.373.2
[26] Bruno Blanchet et al. 2001. An efficient cryptographic protocol verifier based on prolog rules.. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, Vol. 1. 82–96.
[27] Moller Bodo, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. In *Security Advisory 21*. 34–58.
[28] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[29] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification*. 463–469. doi:10.1007/978-3-642-22110-1_37
[30] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. 2021. Validation of Side-Channel Models via Observation Refinement. In *MICRO*. doi:10.1145/3466752.3480130
[31] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[32] David Cadé and Bruno Blanchet. 2012. From computationally-proved protocol specifications to implementations. In *2012 Seventh International Conference on Availability, Reliability and Security*. IEEE, 65–74.
[33] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 666–680. doi:10.1109/SP46214.2022.9833707
[34] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. arXiv:1910.01755 [cs]
[35] Sagar Chaki and Anupam Datta. 2009. ASPIER: An automated framework for verifying security protocol implementations. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 172–185.
[36] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. 2022. SAPIC+: protocol verifiers of the world, unite!. In *USENIX Security Symposium (USENIX Security), 2022*.

[37] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. 2018. DEEPSEC: deciding equivalence properties in security protocols theory and practice. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 529–546.

[38] Tom Chothia and Vitaliy Smirnov. 2010. A traceability attack against e-passports. In *International Conference on Financial Cryptography and Data Security*. Springer, 20–34.

[39] Cas Cremers, Alexander Dax, and Aurora Naska. 2024. Breaking and Provably Restoring Authentication: A Formal Analysis of SPDM 1.2 Including Cross-Protocol Attacks. *IACR Cryptol. ePrint Arch.* (2024), 2047.

[40] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1773–1788. doi:10.1145/3133956.3134063

[41] Cas Cremers, Charlie Jacomme, and Aurora Naska. 2023. Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1235–1252.

[42] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. 2020. A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1–17.

[43] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, Virtual. doi:10.14722/ndss.2021.24286

[44] AP David. 2021. *Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems*. Packt Publishing Ltd.

[45] HOL development team. 2022. HOL Interactive Theorem Prover. https://hol-theorem-prover.org/

[46] Adel Djoudi and Sébastien Bardin. 2015. BINSEC: Binary Code Analysis with Low-Level Regions. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag, Berlin, Heidelberg, 212–217. doi:10.1007/978-3-662-46681-0_17

[47] Danny Dolev and Andrew Chi-Chih Yao. 1981. On the Security of Public Key Protocols (Extended Abstract). In *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*. IEEE Computer Society, 350–357. doi:10.1109/SFCS.1981.32

[48] PKI Task Force. 2004. PKI for machine readable travel documents offering ICC read-only access. *Technical report, International Civil Aviation Organization* (2004).

[49] Anthony Fox. 2019. L3: a specification language for instruction set architectures. https://acjf3.github.io/l3

[50] Jean Goubault-Larrecq and Fabrice Parrennes. 2005. Cryptographic protocol analysis on real C code. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 363–379.

[51] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, Cham, 279–299.

[52] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1853–1869.

[53] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1868–1883.

[54] Gerhard P Hancke. 2011. Practical eavesdropping and skimming attacks on high-frequency RFID tokens. *Journal of Computer Security* 19, 2 (2011), 259–288.

[55] Signal Inc. Latest Version Updated on 2025. *The Signal Specifications*. Technical Report. https://signal.org/docs/

[56] Jan Jürjens. 2009. Automated security verification for crypto protocol implementations: Verifying the jessie project. *Electronic Notes in Theoretical Computer Science* 250, 1 (2009), 123–136.

[57] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin. 2019. Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[58] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[59] Steve Kremer and Robert Künnemann. 2016. Automated analysis of security protocols with global state. *Journal of Computer Security* 24, 5 (2016), 583–616.

[60] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

[61] Andreas Lindner, Roberto Guanciale, and Mads Dam. 2023. Proof-Producing Symbolic Execution for Binary Code Verification. arXiv:2304.08848 [cs.PL] https://arxiv.org/abs/2304.08848

[62] Andreas Lindner, Roberto Guanciale, and Roberto Metere. 2019. TrABin: Trustworthy analyses of binaries. *Sci. Comput. Program.* 174 (2019), 72–89. doi:10.1016/j.scico.2019.01.001

[63] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[64] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2022. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* 54, 6 (2022), 122:1–122:37. doi:10.1145/3456629

[65] Gavin Lowe. 1995. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Inform. Process. Lett.* 56, 3 (Nov. 1995), 131–133. doi:10.1016/0020-0190(95)00144-2

[66] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[67] John McLean. 1992. Proving Noninterference and Functional Correctness Using Traces. *J. Comput. Secur.* 1, 1 (1992), 37–58. doi:10.3233/JCS-1992-1103

[68] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 696–701.

[69] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith (Eds.). Vol. 8044. Springer Berlin Heidelberg, Berlin, Heidelberg, 696–701. doi:10.1007/978-3-642-39799-8_48

[70] Carlo Meijer, Veelasha Moonsamy, and Jos Wetzels. 2021. Where's crypto?: Automated identification and classification of proprietary cryptographic primitives in binary code. In *30th USENIX Security Symposium (USENIX Security 21)*. 555–572.

[71] Nicholas Mosier, Hamed Nemati, John C. Mitchell, and Caroline Trippel. 2024. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 4200–4219. doi:10.1109/SP54263.2024.00048

[72] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1433–1450. https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[73] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. 2023. CryptoBap: A Binary Analysis Platform for Cryptographic Protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. 1362–1376. https://doi.org/10.1145/3576915.3623090

[74] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. 2025. Symbolic Parallel Composition for Multi-language Protocol Verification. *to appear in the 38th IEEE Computer Security Foundations Symposium (CSF)* (2025). https://arxiv.org/abs/2504.06833

[75] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. 2026. Automated Side-Channel Analysis of Cryptographic Protocol Implementations-Source Code. https://github.com/FMSecure/CryptoBAP/tree/ccs2026

[76] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of Abstract Side-Channel Models for Computer Architectures. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–248.

[77] Michael Neve and Jean-Pierre Seifert. 2007. Advances on Access-driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography* (Montreal, Canada) *(SAC'06)*. Springer-Verlag, 147–162.

[78] Santiago Arranz Olmos, Gilles Barthe, Chitchanok Chuengsatiansup, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. 2024. Protecting cryptographic code against Spectre-RSB. *IACR Cryptol. ePrint Arch.* (2024), 1070. https://eprint.iacr.org/2024/1070

[79] Kentrell Owens, Anita Alem, Franziska Roesner, and Tadayoshi Kohno. 2022. Electronic Monitoring Smartphone Apps: An Analysis of Risks from Technical, Human-Centered, and Legal Perspectives. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4077–4094. https://www.usenix.org/conference/usenixsecurity22/presentation/owens

[80] Nicholas O'Shea. 2008. Using Elyjah to analyse Java implementations of cryptographic protocols. In *Joint Workshop on Foundations of Computer Security,*

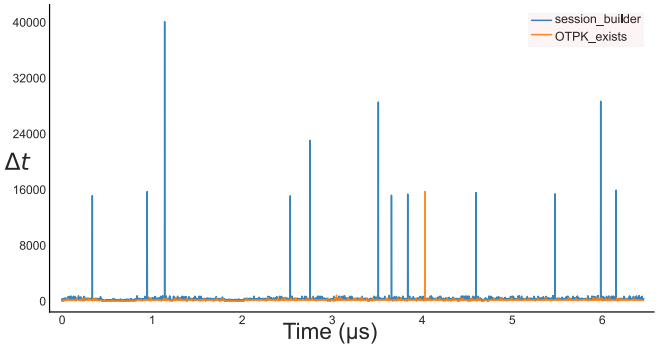

Figure 8: The observed access latency (Δt) during the designated time period. *session_builder* represents the address of the function that initiates a secure session, while *OTPK_exists* indicates the address of an instruction executed when a one-time pre-key is used in the creation of the master secret key for the secure session.

Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS-2008).

[81] Lawrence C. Paulson. 1998. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security* 6, 1-2 (Jan. 1998), 85–128. doi:10.3233/JCS-1998-61-205

[82] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.

[83] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1753–1770. doi:10.1109/SP46215.2023.10179355

[84] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1094–1111. doi:10.1109/SP46215.2023.10179418

[85] Huzaifa Sidhpurwala. 2019. Security flaws caused by compiler optimizations. *Red Hat Blog* (2019).

[86] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1–15.

[87] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 2 (Jan. 2010), 37–71.

[88] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES'03, LNCS)*. Springer, 62–76.

[89] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium*, Michael D. Bailey and Rachel Greenstadt (Eds.). 1415–1432. https://www.usenix.org/conference/usenixsecurity21/presentation/weber

[90] WhatsApp Inc. / Meta Platforms. Version 8 Updated August 19, 2024. *WhatsApp Encryption Overview*. Technical Report.

A Appendix

A. Proof-of-Concept Implementation

To evaluate the feasibility of our attack in Section 4.2.3, we implemented a proof-of-concept targeting the WhatsApp Desktop application. This implementation is based on the Flush+Flush framework [51], modified to be compatible with macOS, and is available at https://github.com/Winona-dev/sca-protocol. Experiments were conducted on a 2019 MacBook Pro equipped with an Intel Core i7-9750H processor (6 cores, 2.6 GHz) and 16 GB of RAM, running macOS Sonoma Version 14.1.2.

The attack leverages a Prime+Probe instruction cache side channel to infer whether the victim initiates a new conversation, specifically, whether a secure session is being established. The adversary controls a co-resident user-space process scheduled on the same physical core as the target application. We assume the attacker knows the virtual addresses of two functions within the WhatsApp binary: one responsible for initiating a secure session, and another triggered when a one-time pre-key is present (*session_builder* and *OTPK_exists* in Figure 8, respectively).

In each attack cycle, the attacker primes the instruction cache with knowledge of these addresses to access another code that maps to these cache sets and lines. After a short delay to allow for potential victim execution, the attacker re-accesses the same addresses and measures the access latency (Δt). Elevated latency indicates eviction, implying that the victim executed instructions mapping to the same cache line. Conversely, low latency suggests the cache lines remained untouched.

Our macOS-compatible iteration of the Flush+Flush framework integrates timing-based cache probing via the *mach_absolute_time()* function to measure time intervals. This function offers time measurements in platform-dependent units and, on macOS, produces a high-resolution timestamp akin to a cycle count. It is employed for precise timing evaluations of access latency. Our implementation employs a straightforward access timing technique to infer potential cache evictions, thus enhancing the accuracy of cache-related timing calculations.

Figure 8 presents a time series plot of the access latency Δt. The x-axis represents time (in microseconds), while the y-axis denotes measured latency. Distinct latency spikes correspond to the victim executing one or both target code paths, revealing instances of first-time conversation establishment (highlighted in red in Figure 8).

B. Ethical Considerations

Our work is intended for defensive security: it extracts protocol-relevant models from executables and analyzes them under explicit microarchitectural leakage contracts to find vulnerabilities.

Human subjects and data. Our analysis of WhatsApp Desktop reveals a side-channel attack that enables inference of the victim's social graph. This information is sensitive personal data, and unauthorized inference could breach users' right to privacy. Therefore, we did not conduct human-subject research and did not collect or analyze personal user data. Experiments were performed offline on locally executed software and on implementations/test harnesses under our control.

Responsible disclosure and artifacts. For newly identified issues, we have followed responsible disclosure with Meta and provided them with results and instructions to reproduce found vulnerabilities. Meta confirmed the existence of the reported vulnerabilities. We have not received confirmation of a deployed fix as of this writing; therefore, some risks remain unmitigated.

Authorization. We analyze only software we can lawfully obtain and do not access systems without authorization. Our experiments involved reverse engineering the WhatsApp Desktop application. Although this activity may conflict with Meta's license agreements, we responsibly disclosed our findings to Meta upon discovery.

Societal impact. The societal impact of analyzing WhatsApp is considerable, as it is used by billions of people for personal and professional communication. By examining WhatsApp's binary implementation rather than its specification, our work helps close the gap between advertised guarantees and the protections users actually receive in practice. Our discovery of an implementation-level side-channel vulnerability raises users' awareness of hidden risks that can threaten privacy, including social graph inference—especially for journalists, activists, and users in repressive environments. Sharing such analyses publicly, with responsible disclosure, fosters societal trust in secure messaging by encouraging vendors to remediate subtle flaws and by supporting higher standards of accountability and thoroughness in the development and testing of widely used cryptographic systems.

C. Open Science

The source code of our framework can be accessed at [75]. The other artifacts necessary to evaluate the core contributions of our paper are available via their respective links provided below.

- CryptoBap: https://github.com/FMSecure/CryptoBAP
- Tamarin (including Sapic$^+$, which is integrated into Tamarin): https://tamarin-prover.com/
- ProVerif: https://bblanche.gitlabpages.inria.fr/proverif/
- DeepSec: https://deepsec-prover.github.io/
- Ghidra: https://github.com/NationalSecurityAgency/ghidra

D. Use of Generative AI and LLMs

This paper has been refined for grammar, spelling, and minor style improvements with Grammarly.