# CauSec: Unboxing the Causal Drivers of Static Vulnerability Analysis Performance

Md Akram Khan, Daniel Rodriguez-Cardenas, Alejandro Velasco Dimate,
Denys Poshyvanyk, and Adwait Nadkarni
*William & Mary*



***Abstract*—Static Application Security Testing (SAST) tools are widely used in both industry and academia. Such tools often make design choices that sacrifice detection to achieve higher performance, *i.e.,* increased precision, decreased runtime, or increased scalability. These design choices rely on certain *assumptions* regarding the target code or the analysis technique itself. Hence, the assumptions directly impact the detection outcome through the design choices they influence. This motivates a key question: do the sacrifices in the detection capabilities actually help tools achieve the expected performance gains? That is, *are the underlying assumptions valid?***

**This paper seeks to address this question by relying on a key observation that the assumptions made by these tools are generally of a *causal* nature. We propose CAUSEC, a causal analysis framework that makes SAST assumptions testable and *explains why* the performance changes given certain assumptions, beyond simple correlations. CAUSEC formalizes the assumptions of the SAST tool into the abstraction of a *security assumption* and combines assumption-driven causal modeling with effect estimation and validation to test its validity and investigate the factors affecting it. To understand what security assumptions generally entail, we perform a systematic literature review of SASTs that detect crypto-API misuse, leading to the discovery and qualitative analysis of 57 assumptions. We then demonstrate the utility and robustness of CAUSEC by testing a popular assumption in four highly relevant tools, using a manually labeled ground truth dataset consisting of $57,038$ alerts. Our analysis leads to several key findings that represent insights regarding assumptions and causal effects, which we distill into $3$ takeaways for future work.**

## 1. Introduction

Static Application Security Testing (SAST) tools play a critical role in preventing vulnerabilities in end-user software, and are widely used both in industry and academia [1], [2], [3]. However, recent work has found that SASTs may be prone to *design and implementation flaws* that result in significant gaps in analysis, which prevent them from detecting the very vulnerabilities they claim to detect [4], [5]. Such gaps can lead to failures in critical use cases such as compliance certification, enabling vulnerable but certified/trusted end-user software [6].

Although some flawed design choices are implicit, *i.e.,* unintentional errors on the part of tool designers, some may be well-intentioned attempts at addressing the trade-off between the detection rate and practical objectives such as precision and scalability. That is, we observe that SASTs often base their design choices on *assumptions* regarding code, *e.g.,* CogniCrypt [7] chooses to not report alerts from certain third-party libraries, as they could lead to increased false positives, while CryptoGuard [8], until recently, excluded alerts from core Android libraries for the same reason [5], [9]. These assumptions directly influence what code is analyzed or what alerts are reported, and hence, clearly impact detection outcomes. This motivates a critical research question: ***are the assumptions that motivate the gap-inducing design choices valid?***

This paper seeks to address this question by contextualizing the approach of *causal inference* [10] to systematically test the assumptions made in SASTs. Our choice is motivated by a key observation: that the assumptions bounding the detection of SASTs are often *causal* in nature. For example, the assumption that "increasing exploration depth does not increase recall" implies a causal relationship between the condition (*i.e.,* increase in exploration depth) and the outcome (*i.e.,* constant or decreased recall). If formulated correctly, such assumptions can be expressed as causal hypotheses and tested via intervention.

We propose the CAUSEC framework for testing assumptions in SASTs using causal inference. CAUSEC proposes the abstraction of a *security assumption* that uniformly captures the preconditions, conditions, and outcomes expressed in diverse assumptions found in SASTs. Beyond standardizing assumptions, CAUSEC reformulates them into operational forms suitable for causal evaluation, preserving their intended meaning while exposing interventions that can be tested empirically. CAUSEC then constructs *treatments* that realize these interventions at appropriate stages of the SAST workflow, to test *what if* a certain condition were to change; *e.g.,* including or suppressing alerts from third-party libraries (data-level treatment), or enabling/disabling a certain exploration depth (method-level treatment). CAUSEC analyzes a causal model that makes explicit all the variables that must be considered when testing the assumption, including the *confounders* that affect both the treatment and the outcome. In the end, CAUSEC produces an adjusted effect estimate under the specified causal model and identification assumptions, while controlling for measured confounders. By making the modeling choices explicit and applying refutation tests, CAUSEC helps assess whether the resulting

estimate remains stable under reasonable perturbations.

While CAUSEC is a general framework for testing SAST assumptions, we focus our examples and evaluation on an important class of SASTs, *i.e.,* crypto-API misuse detectors or *crypto detectors* in short. To understand the landscape of assumptions in this domain, we first perform a qualitative study of assumptions in crypto detectors. We then empirically evaluate CAUSEC using a popular assumption and $57,038$ alerts from 4 popular crypto detectors, Semgrep, CodeQL, CryptoGuard and CogniCrypt. Our evaluation leads to 10 key findings that provide insight into (1) *what* assumptions are, (2) *how* they can be operationalized and evaluated, and (3) *how* the assumption holds (or does not) across tools and contexts. This paper makes the following contributions along these dimensions:

- **Characterizing Security Assumptions (the *what*) –** We perform a systematic literature review (SLR) focused on crypto detectors *from the last 20 years*, which leads to the identification of 57 unique assumptions regarding diverse outcomes (*e.g.,* precision, recall, runtime, scalability). We qualitatively analyze these 57 assumptions to uncover latent insights and highlight useful patterns.
- **The CAUSEC Framework (the *how*) –** We design the CAUSEC framework that contextualizes causal inference to evaluate SAST assumptions under explicit structural causal models. CAUSEC formalizes the notion of a security assumption, enabling informal assumptions to be reformulated into operational forms suitable for causal evaluation. CAUSEC provides mechanisms for constructing treatments, identifying causal estimands, estimating causal effects, and evaluating the robustness of the resulting estimates.
- **Evaluation with SASTs (the *why*) –** We demonstrate the utility of CAUSEC by evaluating a popular assumption *i.e.,* that reporting alerts from third-party libraries decreases precision, using four popular SASTs, Semgrep, CodeQL, CogniCrypt and CryptoGuard. We analyze a stratified representative sample of 558 applications with the tools, and develop a manually validated ground truth dataset of $57,038$ alerts. We not only evaluate the adjusted effect of reporting alerts from third-party library but also estimate how this effect varies across library categories while adjusting for measured confounders. We have anonymously released our artifact [11].

Our analysis of assumptions and evaluation of CAUSEC leads to 10 insightful findings ($\mathcal{F}_1 - \mathcal{F}_{10}$) about the assumptions and the causal effects estimated by CAUSEC, which we distill into 3 discussion themes and corresponding takeaways for future research. In particular, our evaluation leads to a key takeaway: *tool design can act as an effect modifier*, such that the estimated causal effect of the same assumption can differ substantially across tools.

## 2. Background

This section establishes the foundational concepts necessary for conducting a rigorous causal analysis on vulnerability analysis performance. We structure our section around the theoretical foundations using Pearl's framework [12], [13], [14] that underpins causal reasoning and the practical methodologies for identifying and estimating causal effects from SAST assumptions as observational data.

**Causal Analysis Fundamentals.** Consider a static analysis tool (SAST) that flags more vulnerabilities in large, complex codebases. One might conclude that code complexity *causes* more vulnerabilities to appear. However, larger projects also tend to have more developers and longer histories, both of which independently increase vulnerability count. This is a classic *confounding* problem: the variable "project size" influences both the treatment (SAST flag rate) and the outcome (vulnerability count), creating a spurious association. Causal inference provides the formal machinery to disentangle such effects.

Modern causal analysis is grounded in *Structural Causal Models* (SCMs) [10], [15], which encode causal relationships as structural equations over system variables. SCMs are visualized as *Directed Acyclic Graphs* (DAGs), where nodes represent variables and directed edges represent direct causal influence. An edge $T \to Y$ means that $T$ (treatment) appears in the mechanism that determines $Y$ (outcome); crucially, this directionality is asymmetric and distinguishes causal mechanisms from mere statistical associations.

A confounder $Z$ is a variable that influences both $T$ and $Y$, introducing spurious associations that corrupt naive estimates of causal effects. Pearl formalizes this tension between *seeing* (observational associations) and *doing* (interventional effects), through *do-calculus* [10], [13], [16]. Confounding is present when the observational and interventional distributions differ *i.e.,* $p(y \mid t) \neq p(y \mid \text{do}(t))$.

To remove confounding bias, Pearl's *backdoor criterion* identifies which variables $Z$ must be conditioned on to block all non-causal paths from $T$ to $Y$ [12], [15], [17]. Formally, $Z$ satisfies the backdoor criterion if it (1) blocks all backdoor paths into $T$, and (2) contains no descendant of $T$. When satisfied, the causal effect is identified by the backdoor adjustment:

$$P(Y \mid \text{do}(T{=}t)) \;=\; \sum_{z} P(Y \mid T{=}t,\, Z{=}z)\; P(Z{=}z).$$

This formula compares treated and control units *within strata* of $Z$, then averages over the marginal distribution of $Z$, yielding an unbiased causal effect estimate.

**Estimand, Estimator, and Estimate.** Quantifying a causal effect requires answering three questions in sequence, (1) *what* quantity should be estimated, (2) *how* it should be computed from data, and (3) *how much* the effect is. These correspond to the *estimand*, the *estimator*, and the *estimate*.

The *estimand* is the target causal quantity, expressed as a mathematical formula derived from the causal assumptions encoded in the DAG. Common estimands include the Average Treatment Effect *ATE*, the Average Treatment Effect on the Treated (*ATT*), and the Conditional Average Treatment Effect *(CATE)*, among others (see Appendix A for a full list). In our study, we focus on the *ATE*, which measures the expected difference in outcomes between treated and control

units across the entire population: $ATE = \mathbb{E}[Y(1) - Y(0)]$, answering: *on average, what is the effect for everyone?*

The *estimator* is the methodological procedure used to compute the estimand from observed data, for example, propensity score matching (PSM), inverse probability weighting (IPW), or regression adjustment.The choice of estimator depends on the causal question, data characteristics, and modeling assumptions. Applying the estimator to the data yields the *estimate*, which is a numerical value of the causal effect accompanied by uncertainty quantifiers such as confidence intervals or standard errors.

Note that once assumptions are formalized in a DAG, *identification methods* determine whether and how causal effects can be estimated from observational data. Pearl's *do*-calculus provides formal rules for this assessment, establishing whether the estimand can be computed from the observed data distribution and causal model assumptions. The causal effects identified successfully can then be estimated using one of practical estimation approaches described previously (*e.g.,* PSM).

## 3. Related Work

CAUSEC builds on the theoretical foundations of causal inference and its adaptations in software engineering, and to our knowledge, it is the first work to contextualize causal inference for the task of evaluating assumptions in security tools. This section describes related work in two key areas:

**Causal Inference in SE:** Causal reasoning is gaining traction in software engineering, with recent work applying Pearl's causal framework, including structural causal models, do-calculus [10], [16], and backdoor adjustment [12], to problems such as interpreting neural code models [15], [18] and studying security and trustworthiness properties of LLM-based code systems [19], [20], [21]. However, prior work generally focuses on estimating causal effects for a specific software engineering problem once the treatment, outcome, and causal model are already defined. In contrast, our work addresses a different challenge: how to transform informal assumptions embedded in security tools into operational causal assumptions that can be systematically evaluated. To this end, we introduce the abstraction of a *security assumption* and the CAUSEC framework for constructing treatments, identifying causal effects, and evaluating their robustness in the context of SAST design assumptions.

**Security Analysis Tools and Evaluations:** A large body of work has developed SASTs for detecting security vulnerabilities, including data leaks [22], [23], [24], [25], [26], crypto-API misuse [2], [8], [27], [28], [29], [30], [31], [32], [33], [34], and permission misuse [35], [36]. At the same time, there is growing recognition that these tools are often soundy in practice [37], making deliberate trade-offs in detection for gains in precision, runtime performance, or scalability.

While SASTs have been evaluated with benchmarks that often accompany the tools [38], [39], this realization has prompted dedicated research on evaluating these tools to uncover detection failures that violate the tools' claims. For example, Bonett et al. [4] developed $\mu$se to evaluate data-leak detectors, while Ami et al. [5] developed MASC to systematically evaluate crypto detectors. These studies demonstrated that detection failures may arise not only from implementation defects, but also from intentional design decisions and the assumptions that motivate them. For example, CryptoGuard ignores Android libraries due to the assumption that reporting alerts in Android libraries would decrease precision. However, Ami et al. found that due to a flawed implementation of this design decision, CryptoGuard accidentally ignored all packages containing "android." or ending in "android", preventing CryptoGuard from analyzing several non-Android classes as well.

Our work is motivated by this line of research but differs in its objective. Rather than evaluating whether a SAST succeeds or fails on a benchmark, we focus on validating the underlying assumptions that drive its design choices. We build CAUSEC, the first framework for systematically formalizing, operationalizing, and evaluating SAST assumptions through causal analysis, enabling researchers to estimate the effects of these assumptions in the context of their tools, while explicitly accounting for confounding factors.

## 4. Design Goals

SAST tools are designed based on assumptions about program contexts or the analysis boundaries. If the assumptions are invalid or overstated, the design may incur loss of detection without any justifiable gains in precision or runtime performance. Thus, we seek to introduce a framework that transforms informal SAST assumptions into operational causal assumptions that can be systematically evaluated, guided by the following design goals:

- $\mathbf{G}_1$ *Actionable Representation of Assumptions.* We need a well-defined abstraction that represents diverse SAST assumptions and captures the information required to reformulate informal assumptions into forms that are amenable to causal evaluation.
- $\mathbf{G}_2$ *Transparent and Reusable Framework.* The framework should make explicit the treatments, variables, and modeling choices involved in evaluating SAST assumptions, and explain their impact on the resulting causal estimates. It should be modular and reproducible so that it can be applied to diverse assumptions and SASTs.
- $\mathbf{G}_3$ *Robust Outcomes.* To reliably evaluate assumptions, the framework should include mechanisms to assess whether its causal estimates remain stable under reasonable perturbations to the data and modeling choices.

## 5. The CAUSEC Framework

This paper introduces the CAUSEC framework, which develops a systematic abstraction to express assumptions and contextualizes causal analysis to test them, guided by design goals $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$, as shown in Figure 1.

CAUSEC starts by compiling the assumptions into a formal representation (see $\mathbf{G}_1$), *i.e.,* the *security assumption*.

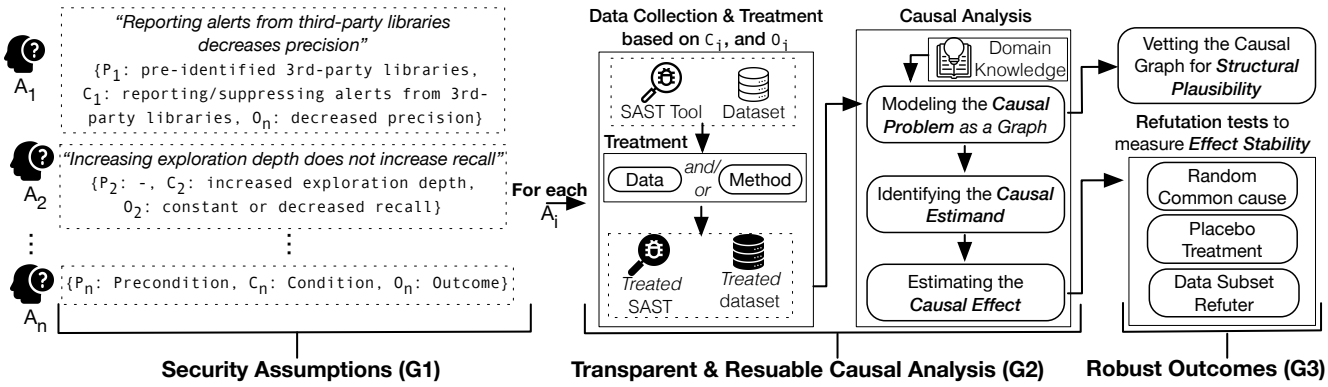


Figure 1: Overview of the CAUSEC Framework

Assumptions are often expressed in ad-hoc and inconsistent form in the documentation associated with SASTs (*e.g.,* readme files, research papers). The *security assumption* provides a precise structure to express assumptions, which consists of three key elements: a precondition, a condition, and an outcome. That is, a security assumption is represented as a causal claim relating a condition (*e.g.,* "reporting alerts from third party libraries") and its impact on the outcome (*e.g.,* "decreases precision"), with the optional precondition specifying any additional setup required for evaluation (*e.g.,* identification of third party libraries). This formulation allows assumptions to be analyzed and compared independently (as we will do in Section 6) and prepares them for systematic testing with CAUSEC (addressing Goal $\mathbf{G}_1$).

We use the formalized assumption to identify key factors and prepare the necessary datasets for causal analysis. In particular, condition ($C_i$) and outcome ($O_i$) help us identify the candidate causal factor, or *treatment*, and how it is operationalized for evaluation. Depending on the condition and outcome being studied, treatments may take the form of changes to the data, SAST methods, or, in some cases, both. In addition, the outcome ($O_i$) helps us identify the measurable behavior for causal analysis, such as precision, recall, or runtime. Section 5.2 describes this process.

Once the relevant data and SAST (s) are prepared, we *model the causal inference problem* as a DAG that shows the relationships among the treatment, the outcome, and the confounding variables, using both domain knowledge and observed data (Section 5.3). CAUSEC then *identifies a causal estimand*, which expresses the effect to be measured under the chosen treatment (Section 5.4). CAUSEC then *estimates the causal effect* based on probabilistic methods that operate on observable data (Section 5.5), which allows us to evaluate whether the empirical evidence supports the assumption under the specified causal model. As the assumptions, data, adjustment variables, and robustness checks used in the causal analysis are made explicit, CAUSEC produces transparent and actionable results, addressing Goal $\mathbf{G}_2$.

Finally, we validate the causal process in two steps (Section 5.6). First, we assess whether the causal graph is consistent with observed relationships among the variables using correlation analysis. Second, we perform refutation tests to check for the stability of the estimated causal effect, *i.e.,* if the estimated effect remains consistent and close in size under reasonable perturbations, we consider the result as stable evidence, addressing Goal $\mathbf{G}_3$. The rest of this section explains the design of CAUSEC in depth, providing both design rationale and details of the process.

## 5.1. The Security Assumption

Assumptions are often stated informally in artifacts and may not directly correspond to an intervention that can be evaluated through causal analysis. For example, consider the following assumption: "Analyzing third-party libraries decreases precision". Such assumptions about analysis behavior (*e.g.,* analyzing third-party libraries) may be ultimately concerned with the outcomes defined over the reported alerts, such as precision or recall. Thus, in addition to standardizing assumptions, we seek to *reformulate them into an operational form suitable for causal evaluation*, which preserves their intended meaning while exposing a condition and outcome that can be operationalized as a treatment. This reformulation also determines *where* the treatment is applied during evaluation, such as at the analysis input, within the analysis process, or post-hoc on the analysis output. For example, the informal assumption "analyzing third-party libraries decreases precision" may be reformulated as "reporting alerts from third-party libraries decreases precision," (shown as $A_1$ in Figure 1) since precision is measured over reported alerts. Thus, the corresponding treatment is applied post-hoc to the analysis output (*i.e.,* reported alerts) rather than to the analysis input (*i.e.,* application code) or the analysis process (*i.e.,* the SAST itself), while preserving the outcome semantics of the original assumption.

To standardize assumptions in a consistent, understandable, and actionable form for causal analysis, we formulate the abstraction of a *security assumption*. This abstraction allows us to distinctly capture the context in which an assumption applies, the condition being asserted, and the outcome of interest, enabling the assumption to be operationalized as a causal treatment and evaluated through causal

analysis. A security assumption ($A_i$) is represented by a tuple consisting of the precondition ($P_i$), the condition ($C_i$), and the outcome ($O_i$), as follows:

$$A_i = (P_i, C_i, O_i)$$

Here, the **precondition** ($P_i$) is the scope or context in which the assumption is applied. It captures any stated or implied prerequisites (*e.g.,* dataset constraints, platform constraints, or analysis configuration) that must be met for the assumption to be meaningful. Not all assumptions may be accompanied by preconditions. The **condition** ($C_i$) is the specific claim, heuristic, or constraint on which the assumption is based. It describes the aspect of the assumption that is expected to influence the outcome and serves as the basis for constructing a causal treatment. The **outcome** ($O_i$) describes the expected consequence once the condition is met, such as the impact on precision, recall, runtime, or scalability. Making the outcome explicit completes the formalization and gives us an observable metric for estimating the causal effect associated with the assumption. Together, the condition $C_i$ and outcome $O_i$ determine how the treatment is constructed for causal evaluation.

As an example, consider an assumption we found in *Cryptolation* [40], a cryptographic API misuse detector for Python, as follows: *"Different from other SCA tools, we identify the imports that we have misuse patterns for and only scan for those. We call this an import-driven approach....**Using an import-driven approach increases performance and does not sacrifice accuracy or precision**."* This example can be formulated into ***3 separate security assumptions***, since the same condition (restricting analysis scope) is associated with three distinct outcomes: improved runtime performance, maintained detection rate, and maintained precision. For all three security assumptions, the **precondition ($P_i$)** would be that the source code contains imports corresponding to cryptographic APIs for which misuse patterns are defined, and the **condition ($C_i$)** would be that the analyzer uses the aforementioned import-driven strategy that restricts scanning and slicing to only the code regions associated with the imports defined in the precondition. However, each of the three assumptions assume a unique **outcome ($O_i$)**, *i.e.,* runtime performance, detection rate, or precision. As a result, although the assumptions share the same condition, they may require different treatments and evaluation procedures depending on the outcome.

## 5.2. Data Collection and Treatment

The abstraction of the security assumption also helps determine the data needed to evaluate a specific assumption, as well as how the treatment should be constructed from the condition and outcome.

For ease of explanation, consider the assumption $A_1$ from Figure 1, *i.e.,* that "*Reporting alerts from third-party libraries decreases precision.*" Intuitively, to test this assumption, we would need to collect the following types of data: (1) a representative set of decompiled apps, (2) identification of code paths in the apps that constitute third-party libraries, (3) alerts from one or more SAST tools (and, in the context of this paper, crypto-detectors) produced from analysis of the apps, including alerts originating from both developer-written and third-party code, and (4) ground-truth labels on all the alerts via manual validation, classifying them as true or false positives.

The reformulation performed during the construction of a security assumption (Section 5.1) determines where the corresponding treatment is applied within the SAST workflow (*i.e.,* the input, analysis process, or output). For $A_1$, the treatment would be *reporting/considering alerts originating from third-party libraries* in addition to developer-origin alerts. As we described in Section 5.1, $A_1$ is operationalized at the analysis output because the outcome of interest, precision, is defined over reported findings. The intervention is thus applied post-hoc to the reported alerts, constituting a change to the alert dataset, *i.e.,* a *data-level* treatment. This treatment avoids repeatedly executing the analyzer under different library configurations while preserving the semantics of the outcome specified by the original assumption.

In contrast, if the outcome $O_1$ were *runtime* rather than precision, *e.g.,* if the assumption were that "analyzing third party libraries significantly increases runtime", we would need a method-level treatment that changes the scope of the analysis. That is, we would need to modify the tool to analyze or skip libraries during execution and use the resulting runtime measurements to estimate whether the observed effect is consistent with the assumption.

## 5.3. Modeling the Causal Analysis Problem

After collecting the dataset, the next step in CAUSEC is to model the security assumption as a causal analysis problem. This step makes explicit the treatment derived from the assumption, the outcome being evaluated, and the variables that may influence both. In the context of SAST evaluation, many factors, such as app size, popularity, category, and dependency usage, are not randomly distributed and may confound the relationship between the treatment and outcome. We therefore represent the causal problem as a directed acyclic graph (DAG) that captures the treatment, outcome, and potential confounding variables. The resulting model provides the foundation for causal identification, effect estimation, and robustness analysis.

**Unit of analysis and outcome:** For our running example, the unit of analysis is an alert. Each alert is a single report produced by an SAST tool on a specific code location within an app. The outcome variable is precision:

- **Outcome ($Y$):** *verdict* $\in \{0, 1\}$, where 1 denotes a true positive (TP) and 0 denotes a false positive (FP).

We treat an alert as correct if manual validation confirms the reported misuse as a true positive, and incorrect otherwise. Consequently, verdict serves as an alert-level representation of precision. The estimated effect should therefore be interpreted as a change in the probability that a reported alert

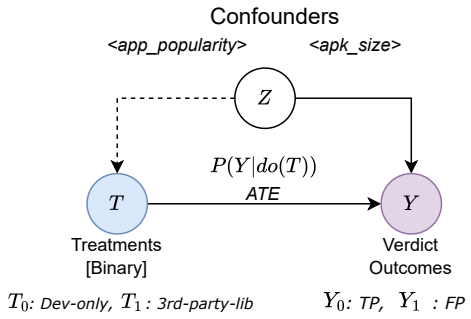


Figure 2: The Structural Causal Model for evaluating $A_1$

is correct, rather than as a deployment-level effect that also considers alert volume, deduplication, or developer triage.

**Treatments and contextual variables.:** CAUSEC supports multiple causal questions by allowing different variables to be chosen for treatment. We use the following variables:

- **Reporting policy:** *3rd-party-lib* $\in \{0, 1\}$, where 1 indicates that alerts from third-party libraries are reported in addition to developer-written alerts, and 0 indicates that only developer-written alerts are reported.
- **App popularity:** *app_popularity*, an ordinal bucket representing the download count of that app on Google Play.
- **App size:** *apk_size*, the size of the APK file.

Depending on the assumption being evaluated, these variables may serve as treatments, confounders, or effect modifiers. The causal model captures how they influence the outcome and how they relate to one another.

**The Structural causal model:** We model these relationships using an SCM (see Section 2). The SCM encodes which variables are assumed to directly influence others, while leaving room for unmeasured background factors. This lets us reason about interventions explicitly by setting a treatment while holding the rest of the system fixed.

For example, for the assumption $A_1$, several contextual variables may influence both the reporting policy and alert correctness. App popularity, measured by download count, can affect dependency usage patterns [41], which in turn influence the prevalence and characteristics of third-party library alerts. Similarly, larger applications often incorporate more third-party code, making app size another potential confounding factor. We encode such relationships, derived from domain knowledge and observable data, into a directed acyclic graph (DAG), referred to as the *causal graph*. One such causal graph for our running example is shown in Figure 2, which encodes the following relationships:

- *app_popularity* $\rightarrow$ *3rd-party-lib*: Popularity influences the likelihood that an alert is located in third-party code (popular apps tend to have more third-party code).
- *app_popularity* $\rightarrow$ *verdict*: Popularity can influence alert correctness through many ecosystem factors (*e.g.,* code organization and dependency practices).
- *apk_size* $\rightarrow$ *3rd-party-lib*: Larger apps are more likely to contain third-party code.
- *apk_size* $\rightarrow$ *verdict*: Larger apps contain more source code, dependencies, and execution contexts, which might make static analysis less precise.
- *3rd-party-lib* $\rightarrow$ *verdict*: The reporting policy can influence alert correctness because alerts originating from third-party libraries may exhibit different true-positive and false-positive characteristics than alerts originating only from developer-written code.

This graph is deliberately compact, as we include only variables that are (i) measurable in our evaluation pipeline, and (ii) plausibly responsible for confounding or heterogeneity in alert correctness based on domain knowledge. As with any observational causal analysis, the graph does not claim to capture all possible causes of alert correctness. Instead, it makes explicit the variables for which adjustment is possible, together with the assumptions under which the estimated causal effect should be interpreted.

## 5.4. Identifying the Causal Estimand

After identifying the variables and expressing their relationships in the causal graph, the next step is to identify the causal estimand, *i.e.,* the causal quantity that captures the effect of the treatment on the outcome. For a fixed causal graph and treatment-outcome pair, the same estimand defines the target causal quantity across datasets [10]. Recall that in our setting, $Y$ represents alert correctness, where $Y = 1$ denotes a true positive and $Y = 0$ denotes a false positive. Because $Y$ is binary, $\mathbb{E}[Y \mid do(\cdot)]$ can be interpreted as the probability that an alert is a true positive under the intervention. Thus, estimated effects can be interpreted as changes in the true positive probability, *i.e.,* precision.

As described in Section 2, several estimands can be used to quantify causal effects, such as ATE, CATE, and ATT. We use ATE as our goal is to evaluate the average effect implied by a security assumption across the overall population under study, *i.e.,* all apps in the dataset.

**From the causal graph to an *identifiable* estimand.** Defining an estimand is not sufficient; it must also be identifiable from observational data. Identification links the interventional quantity $P(Y = 1|do(T = t))$ to an expression that can be estimated from observed data. The causal graph plays a central role here by making potential confounding relationships between the treatment $T$ and the outcome $Y$ explicit.

As described in Section 2, there are several strategies to identify the estimand, such as the front-door adjustment, the instrumental variable (IV) and the backdoor criterion. Of the identification strategies described in Section 2, CAUSEC uses the backdoor criterion because its treatments are observational rather than randomly assigned. For example, code provenance, popularity tier, or library family may share common causes with alert correctness. The backdoor criterion identifies an adjustment set that blocks these non-causal paths, allowing the interventional effect to be estimated from observed data under the specified causal graph.

### 5.5. Estimating the Causal Effect

After the estimand is identified, CAUSEC estimates the causal effect from the data using statistical and machine learning methods. The goal is not to build a predictive model, but to estimate the interventional quality specified by the estimand. To do so, treated and control observations are compared only after adjusting for confounding variables identified from the causal graph, *i.e.,* variables that affect both the assignment of treatment and the outcome.

In our case study, treatments are binary, and therefore CAUSEC uses propensity score matching (PSM) to estimate the causal effect (see Appendix E.1). In the evaluation, we additionally apply logistic regression adjustment as an alternative estimator to assess whether the direction and magnitude of the estimated effect remain consistent across matching-based and model-based approaches.

For our causal question, $Y$ is binary (*verdict* $\in \{0, 1\}$), so the ATE is directly interpretable as a change in the probability that an alert is a true positive. Formally, the ATE compares the expected outcome in the treated setting versus the control setting [15], as follows:

$$\text{ATE} = \mathbb{E}[Y \mid do(T = 1)] - \mathbb{E}[Y \mid do(T = 0)].$$

For example, an ATE of $-0.23$ would mean that, under the model assumptions and after adjustment for confounders, alerts in the treated condition (*e.g.,* considering alerts from third-party libraries) are about 23 percentage points less likely to be true positives than alerts in the control condition.

### 5.6. Robustness of CAUSEC's Outcomes

The final stage of CAUSEC assesses robustness at two levels. First, we check whether the causal graph aligns with basic empirical patterns in the observed data. Second, we test whether the estimated effect remains stable under perturbations that should either preserve the effect or nullify it. These checks help CAUSEC identify estimates that are stable and therefore credible under the causal model.

**Vetting the causal graph using empirical checks:** We first assess the causal graph for *structural plausibility* by comparing key relationships implied by the graph against observable data. For example, for edges in the graph that were suggested by domain knowledge, *e.g.,* such as app popularity or app size influencing the treatment in Figure 2, we calculate simple association measures (*e.g.,* Spearman correlation) using the application and labeled alert datasets prepared for causal estimation.These checks do not establish causality, but they help identify graph assumptions that are inconsistent with the observed data.

**Evaluating robustness of the estimated effect:** Refutation tests perturb the data or variables to assess *effect stability*. That is, the goal is not to prove causality, but to check whether the estimate is stable or overly sensitive to changes that should not affect a valid causal relationship. CAUSEC uses the following refuters to assess the stability of the causal effect estimate:

**$\mathbf{R}_1$ Random common cause.** An independent random variable $R$ is added to the model as an additional common cause. Because $R$ is unrelated to the treatment and outcome, a stable estimate should remain essentially unchanged:

$$p(Y \mid do(T)) \approx p(Y \mid do(T), R).$$

**$\mathbf{R}_2$ Placebo treatment.** The original treatment is replaced with an independent placebo treatment of the same type, while the rest of the data remains fixed. Since the placebo treatment should have no real causal connection to the outcome, the estimated effect should move close to zero.

**$\mathbf{R}_3$ Data subset refuter.** The effect is re-estimated on randomly selected subsets of the data. A stable estimate should remain directionally consistent and close to the original estimate across these subsets.

**$\mathbf{R}_4$ Dummy outcome refuter.** The original outcome is replaced with a synthetic one generated to have no causal dependence on the treatment. A stable estimator should produce an effect close to zero for this dummy outcome.

For refutation tests, a small p-value indicates that the refuted estimate differs significantly from the original estimate, suggesting sensitivity to that perturbation rather than stronger evidence for the original effect.

## 6. Study of Assumptions in Crypto Detectors

CAUSEC is motivated by the assumptions in SASTs that influence their design decisions. Extracting such assumptions from existing literature is essential to formulate them as security assumptions in CAUSEC, so that they can be properly tested. This section describes our methodology for extracting assumptions related to an important type of SAST, crypto-API misuse detectors, or *crypto detectors*, as well as the salient findings obtained from their analysis.

### 6.1. Methodology

Security assumptions influence the design choices made by SAST tools, and are hence, very likely to be embedded in the literature describing SASTs, particularly research papers from the security and software engineering communities that propose SASTs. Therefore, we leverage the systematic literature review (SLR) approach as our methodology for extracting assumptions, guided by standard best practices and recommendations by Kitchenham et al. [42]. Further, we qualitatively analyze the assumptions to uncover the latent insights and design rationale embedded in them.

**1. Identifying the information sources:** We considered the proceedings of top-tier venues in security and software engineering (*e.g.,* USENIX Security, CCS, S&P, ACSAC, NDSS, ICSE, ASE, FSE), published between 2005-2025. In addition, we also performed a thorough search for relevant keywords in digital libraries, *i.e.,* the ACM Digital Library, IEEE Explore, and Google Scholar, which helped us identify artifacts that may have fallen outside the top conferences. Appendix B.1 provides the search query.

**2. Inclusion and exclusion criteria:** Our initial search led to a list of approximately 3000 papers from all sources. To decide whether to consider a paper for further analysis, we devised a simple set of ***inclusion criteria***, namely that the paper is published in the last 20 years, and proposes a crypto API misuse detection tool or, at the least, has crypto API misuse detection as a partial focus. We also devised a set of ***exclusion criteria***, namely that the paper is focused on dynamic analysis, was published before 2005, or does not contain any relevant information. Following this methodology, we short-listed 20 papers for extracting assumptions (list in the artifact [11]).

**3. Extracting Assumptions:** We extracted assumptions from the finalized papers using a single-coder approach, consistent with the prior studies [43], [44]. An author read each paper completely and identified the assumptions, which included assumptions regarding the threat model, program modeling choices, analysis scope, coverage boundaries, and general precision, recall, and performance trade-offs. The author recorded each assumption as a unique entry upon discovery and further analyzed other sections in each article, *i.e.,* other than where the assumption was originally found, to ensure correctness and consistency. The broader list of assumptions was also gradually improved during the process to ensure consistency and remove duplicates.

To improve reliability, the extraction process included ongoing team feedback. The coder periodically shared the evolving assumption list with the team, and a second researcher reviewed the extracted assumptions for clarity and consistency. For each assumption, we recorded metadata including the exact quote, and its location in the paper. Through this process, we obtain 57 unique assumptions.

**4. Formalization into Security Assumptions:** We formalized each extracted SAST assumption using the security assumption abstraction that lies at the core of CAUSEC, as described in Section 5.1. We considered mapped the extracted text into the tuple ($A_i$) to obtain the precondition ($P_i$), condition ($C_i$) and the result ($O_i$). When necessary, we first reformulated assumptions into operational forms suitable for causal evaluation, following the methodology described in Section 5.1. For example, assumptions expressed in terms of analysis behavior may be reformulated in terms of observable outcomes when this preserves the intended meaning and enables causal evaluation. We performed consistency checks to avoid duplicates and overly broad statements while maintaining a direct link back to the original quote for every tuple. These formalized security assumptions serve as the input for our qualitative analysis.

**5. Qualitative Analysis:** We used a single-coder qualitative coding approach to identify patterns across assumptions. Our coding followed an iterative development process. We started by creating a preliminary codebook based on our formalized assumptions, including the conditions and outcomes encoded in them (*e.g.,* libraries, precision, recall), as well as the design choices and performance claims often discussed in static analysis tools (*e.g.,* program slicing, taint tracking, analysis depth). This initial codebook served as a foundation for our analysis, but evolved as we coded new assumptions in the data. After coding all the data, we iterated through the codes to uncover insights, which we discuss next.

## 6.2. Assumption Analysis Results

This section describes the results of our analysis of all 57 security assumptions, which we characterize along 5 key themes. We support key observations with representative quotes from the studied papers and the associated assumption numbers ($\mathcal{A}_i$). Our artifact [11] lists the security assumptions along with the quotes and sources.

**1. Strong claims regarding the causal effect of rules on detection accuracy:** While it is unsurprising that most if not all crypto detectors are rule-based systems, we observed that rule coverage is seen as the main factor that affects detection accuracy, which drives detector design as well. For instance, several detectors focus on improving the ruleset as the primary approach to improving accuracy, *e.g.,* as evidenced by this statement in CryptoGo [45]: "*To improve detection accuracy, we derive 12 cryptographic rules ...*" ($\mathcal{A}_2$). This statement illustrates a security assumption about the ruleset (the condition) affecting accuracy (the outcome).

Similarly, we observed tools making assumptions about the impact of compliance with their ruleset on the security of the analyzed software *e.g.,"We list the seven security rules that are used in our work, and* ***any application*** *that violates any of those rules* ***cannot be secure****"*($\mathcal{A}_{39}$). This instance highlights a particularly strong security assumption regarding the relationship between the ruleset and precision, *i.e.,* for the assumption to hold, certain rules would have to ensure 100% precision, as the software "cannot be secure" if they are violated. Moreover, detectors with different rulesets make similar security assumptions regarding the impact of the rules on precision and accuracy, *e.g.,* the detector with 7 rules above [46] considers them sufficient for absolute precision, while another detector with 12 rules [45] claims the same. This introduces ambiguity regarding the real relationship between rules and detection accuracy, motivating CAUSEC's approach of evaluating such relationships.

**Finding 1** ($\mathcal{F}_1$) **–** Detectors **claim rules to be the main driver of detection accuracy**, with strong causal assumptions about their impact on recall and precision.

Since the notion of correctness is based on rules, detection performance can be sensitive to how the rules are configured. We also find strong assumptions along these lines. For instance, Semgrep* [47] (an enhancement to Semgrep [48]) states that *"By editing the configuration of Semgrep, we were able to increase its detection rate from 15.3% (26/170) to 44.7% (76/170)... Configuring a single tool offers better detection rates and produces fewer warnings than using a suite of tools, as it does not produce duplicate warnings between tools."* ($\mathcal{A}_{15}$). Here, Semgrep* makes a strong causal claim, that the causal effect of rule

configuration on detection accuracy is *stronger than* that of "using multiple tools".

**Finding 2 ($\mathcal{F}_2$) –** Tools may make strong causal claims regarding deployment alternatives based on experimentally observed correlations, *e.g.,* claiming that a specific reconfigured tool has a greater impact on the detection accuracy than an ensemble of others. Such claims are highly likely to influence tool design and deployment.

**2. Focus on Precision-engineering:** We observe that tools generally claim that *precision* can be improved through deliberate design choices that suppress noise or retain semantic context. For instance, a recurring design idea is that static code analysis generates too much noise to be directly useful, so crypto-API misuse detectors incorporate pruning and refinement into the main workflow. For example, CryptoGuard [8] makes two assumptions to this effect, *i.e., "CryptoGuard addresses the false positive problem with a set of refinement algorithms derived from empirical observations of common programming idioms and language restrictions"* ($\mathcal{A}_7$) and *"Our detection streamlines the analysis by only analyzing what is necessary...Only after we slice through the import usage do we verify if there is a cryptographic misuse."* ($\mathcal{A}_{12}$). The same reasoning applies in slicing-based systems, where irrelevant statements are filtered out to reduce noise, as seen in FireBugs [49], *i.e., "FireBugs performs intra-procedural program slicing to only include statements related to the API method of interest..."* ($\mathcal{A}_{44}$). These cases indicate strong security assumptions about the impact of static analysis approaches for pruning and refining on the precision of the analysis.

In addition, we find that precision is also assumed to be affected by how well the detector understands and maintains semantic context during analysis. Several assumptions suggest that warnings can be misleading when context is absent, such as when interprocedural effects are ignored. This leads to *functional* false positives, where a pattern appears to indicate misuse but is not actually exploitable. As one detector aptly states: *"We show that without a proper contextual knowledge, the static program slicing approach, employed by...suffers from a significant ratio of functional false positives: a false positive that meets the formal definition of misuse, yet does not introduce an exploitable vulnerability."* ($\mathcal{A}_{26}$). These are strong security assumptions that CAUSEC is equipped to test.

**Finding 3 ($\mathcal{F}_3$) –** Detectors treat pruning as an essential feature for reducing noise, and claim its strong causal effect on precision. Moreover, detectors also assume a strong association between retaining context and higher precision, considering "functional" false positives.

**3. Trading accuracy for performance and scalability:** We find that tools repeatedly frame a broader goal of improving recall and detection accuracy, and yet, often reject or weaken it to avoid performance loss. For example, as stated in Cryptolation [40], *"We discussed and decided against creating a super control flow graph that incorporates all of the files in the project. While this approach would improve the recall and accuracy by identifying potential cryptographic API misuses across files, we did not want to decrease performance."* ($\mathcal{A}_{10}$). Instead, we observe examples of scope-restricting that assume an impact on runtime, recall, and precision, such as per-file filtering, *e.g., "our analysis works upon a per-file basis...reduced our set..."* ($\mathcal{A}_{34}$). Detectors admit that regions of code excluded from analysis can cause misses; however, this is recognized as a valid tradeoff in return for performance, and in some cases, precision.

**Finding 4 ($\mathcal{F}_4$) –** Detectors deliberately trade analysis coverage for scalability, with heuristic-based strategies for defining the reduced scope, and **unverified causal assumptions regarding expected performance gains**.

**4. Assumptions of Actionability:** The usefulness of the tool depends on whether the findings are actionable and whether the outcomes align with developer priorities. Tool designers recognize this aspect, as we find several assumptions regarding a causal effect of certain design choices on actionability. Particularly, some tool designers believe that if tools go beyond detection and show developers how to fix the vulnerabilities, their output would be more useful/actionable. For example, *"FireBugs reports feedback to developers describing what is wrong with the implementation and how to fix it by showing correct usages of crypto APIs. ... integrate cryptographic components correctly and securely since they are often not cryptographic experts."* ($\mathcal{A}_{46}$). Similarly, tools also make assumptions about actionability that motivate them to implement additional techniques to diagnose and report the root cause of misuse ($\mathcal{A}_{40}$).

Besides value additions such as suggesting fixes or root cause analysis, we observe that assumptions regarding developer preferences also drive tools toward lowering the bar for detection. We observe that several tools assume that developers prefer an output with lower false positives, even at the cost of missing vulnerabilities, *e.g.,* as one tool states, *"...we have learned that developers greatly prefer approaches that indicate actual vulnerabilities rapidly and with high precision over approaches that are sound but suffer from false positives and lack efficiency"* ($\mathcal{A}_{57}$).

The assumptions regarding usefulness, actionability, and developer preferences are indeed causal in nature and can be formulated in the form of security assumptions containing design or deployment choices as conditions, and measurable results in the form of usefulness, developer satisfaction, or developer productivity.

**Finding 5 ($\mathcal{F}_5$) –** Detectors assume that certain design add-ons (*e.g.,* incorporating fixes along with alerts) or optimizations (*e.g.,* prioritizing precision over recall) may have a significant causal effect on actionability and developer happiness. Testing these assumptions is challenging, as **they need to be measured in the field**.

**5. Evaluation shapes the performance claims:** Finally, we

observe that performance claims in tools can be misleading without clear definitions of the coverage and ground truth of the tool. To elaborate, we find assumptions supported by evidence that may highly dependent on the chosen scope of evaluation, *e.g.,* selecting a specific set of popular libraries, as we see in this case: *"In practice, the filter utilizes the API data extracted from seven widely used open-source C/C++ cryptography libraries: libcrypto [42], libcrypt [29], cryptlib [4], LibTomCrypt [7], libgcrypt [6], wolfcrypt [9], and Nettle [8]. These libraries have covered nearly 100% usage cases in our firmware dataset."* ($\mathcal{A}_{17}$)

Similarly, incomplete ground truth data can also negatively impact the validity of the evaluation. For example, as stated in Seader [46], *"... the ground truth is incomplete, as it labels some instead of all invocations of Random(). We currently consider the extra calls of Random() found by Seader to be false positives, although the actual precision is higher."* ($\mathcal{A}_{37}$)

**Finding 6 ($\mathcal{F}_6$) –** The measurements of precision, recall, and coverage in detectors may reflect how the test benchmarks were built, and **do not sufficiently establish the causal effect** of the design decisions on the outcomes.

## 7. Evaluation of CAUSEC

We now demonstrate the utility of CAUSEC by testing an assumption. For this study to be both impactful and practical, it is essential to select an assumption that (1) is commonly held across many SASTs, and which can be tested (2) empirically (*i.e.,* avoiding assumptions related to developer satisfaction or behavior), and (3) can be operationalized without significant re-engineering of existing SASTs. Thus, we select the canonical assumption that is commonly seen in our study (Section 6): that *"reporting alerts from third-party library code decreases precision"*. Our evaluation is guided by three evaluation questions (**EQs**):

**EQ1: Does reporting alerts from 3rd party libraries have a causal effect on precision?** This is the primary assumption we seek to evaluate.

**EQ2: Do different types of 3rd-party libraries have different causal impacts on the precision?** Answering this question allows us to quantify how the effect varies across library categories.

**EQ3: Are CAUSEC's causal estimates robust?** This question evaluates whether the estimated effects remain stable under perturbations.

Through evaluating **EQ1**–**EQ3**, we demonstrate the utility of CAUSEC on a highly relevant SAST assumption while generating insights into both the assumption itself and the tools designed around it. We now describe our experimental setup, followed by the results of our causal analysis.

### 7.1. Experimental Setup

We implemented the CAUSEC using the *do-why* library [50], which supports causal analysis from a well-structured dataset (see Appendix D for details).

**1. App sampling:** For a reliable analysis, we obtained a stratified sample of APKs to analyze for crypto-API misuse, balanced across 11 popularity strata as given by their install count (*i.e.,* 100-500, 500-1k, 1k-5k, ...1M-5M, >5M installs). To elaborate, we downloaded APKs from AndroZoo [51] in June 2025, and decompiled them using jadx (version 1.5.2). To obtain a stratified sample of approximately 50 APKs per population strata, we sampled and decompiled about 2,500 APKs (sampling for each stratum in parallel), manually validating and removing those that failed to decompile or were obfuscated. Our final dataset contained 558 APKs spanning the 11 popularity strata.

**2. Ground Truth Alert Dataset:** We analyzed the decompiled APKs using four popular SAST tools, namely Semgrep [48], CodeQL [52], CogniCrypt [7] and CryptoGuard [8]. The tools were chosen based on popularity, relevance to industry and academia, and causal statements regarding third-party libraries. After removing alerts from partially decompiled or obfuscated code, we obtained $57,038$ alerts ($14,590$ from Semgrep, $21,501$ from CodeQL, $5,675$ from CogniCrypt and $15,272$ from CryptoGuard). One author manually validated these alerts to label them as true or false positives, using the alert description, the offending line of code, and the surrounding source-code context, following a standard validation approach for large-scale studies in this area [43], [44]. This process, which took approximately 4.5 person-months to complete, resulted in a **ground-truth dataset of 57,038 labeled alerts**.

**3. Identification of the Causal Graph:** We identified the causal graph iteratively, starting from the simplest plausible specification and then adding candidate adjustment variables based on domain knowledge and empirical sensitivity checks. We considered three candidate variables: app popularity, APK size, and rule identifier (`rule_id`). App popularity and APK size are plausible confounders, *i.e.,* can reasonably affect the treatment assignment and the outcome, as discussed previously in Section 5.3. However, we treat `rule_id` differently. From domain knowledge, the rule that triggers an alert can directly affect the verdict, since broad rules may produce more false positives while more specific rules may produce higher precision. However, `rule_id` does not determine whether an alert is in developer-written or third-party library code. Thus, we model `rule_id` as a factor that may explain rule-specific differences in precision, *i.e.,* an *effect modifier*, but not as a confounder.

We evaluated a sequence of candidate graphs, with all combinations of confounders (*i.e.,* only app popularity, only apk size, or both), and with and without `rule_id` as the effect modifier. In all cases, adding `rule_id` did not change the estimated effect. This indicates that the final estimate is not driven by rule-specific heterogeneity, even though `rule_id` may still be associated/correlated with alert precision.

Therefore, the final graph blocks the main plausible backdoor paths while keeping `rule_id` out of the adjustment set, as shown in Figure 2. We use this graph consistently for our causal estimation.

## 7.2. Results

As a descriptive baseline before any adjustments, we report the raw precision of alerts overall and by alert source in Table 1 in Appendix C. We observe that including library-code alerts lowers raw precision for CodeQL, CogniCrypt, and CryptoGuard: from 0.95 to 0.89 for CodeQL, from 0.70 to 0.48 for CogniCrypt, and from 0.51 to 0.48 for CryptoGuard. In contrast, Semgrep precision increases from 0.55 on alerts from developer-written code to 0.67 when library-code alerts are included. As the datasets are highly imbalanced, with fewer alerts from developer-written code as opposed to those in third-party libraries, we present two simple balancing methods for all the four datasets: (A) downsampling third-party alerts to match the number of developer-written alerts and (B) bootstrapping developer-written alerts to equal the number of third-party alerts. Both methods show the same difference seen in the unbalanced case. We use these raw and balanced precision summaries as observational baselines.

We now describe the results of our causal analysis with respect to **EQ1**–**EQ3**.

**1. Causal effect of reporting alerts from third-party libraries (EQ1):** For CogniCrypt, based on our causal graph and the backdoor criterion, the estimated ATE is $-0.283$, with a confidence interval of $[-0.381, -0.225]$. This means that, compared to only reporting developer-origin alerts, reporting third-party alerts as well reduces the probability that an alert is a true positive by 28.3 percentage points, holding the confounding factors constant.

**Finding 7 ($\mathcal{F}_7$) –** Reporting alerts from third-party code can lead to an estimated precision drop of $0.283$ in CogniCrypt, after controlling for confounders. This causal estimate is higher than the $0.216$ (about 22%) decrease observed in the raw precision numbers. That is, **the assumption not only holds** for CogniCrypt, but **the adjusted effect is substantially larger** than the effect suggested by the unadjusted observations in a one-off experiment.

For CodeQL, we observe a smaller but still negative causal effect, *i.e.,* the ATE is $-0.057$, with a confidence interval of $[-0.072, -0.017]$. Thus, including third-party library alerts decreases CodeQL precision by about $5.7$ percentage points after adjustment.

In contrast, CryptoGuard and Semgrep show positive estimated effects. For CryptoGuard, the ATE is $0.043$, with a confidence interval of $[0.018, 0.150]$, suggesting a small precision increase of about $4.3$ percentage points. For Semgrep, the ATE is $0.109$, with a confidence interval of $[0.107, 0.202]$, suggesting a precision increase of about $10.9$ percentage points.

**Finding 8 ($\mathcal{F}_8$) –** The assumption that reporting third-party library alerts decreases precision holds for CogniCrypt and CodeQL, but not for CryptoGuard and Semgrep. That is, **tool design acts as an effect modifier**: the same reporting policy can produce different causal effects across tools because of differences in rule design, alert filtering, and implementation choices.

**2. Causal effect across library categories (EQ2):** We cluster libraries into four types using LibScout [53], *i.e.,* utilities, Android, Small libraries (combining the Social Media, Analytics, Advertising, and Cloud types as they had too few alerts each), and miscellaneous (which could not be classified by LibScout). The treatment remains the same, *i.e.,* that of *including/reporting* alerts from libraries of a specific type, along with developer-written code, to estimate the causal effect.

We find that the effect of third-party libraries on precision is not uniform: both the direction and magnitude of the effect vary across library families and SASTs. For CogniCrypt, Utilities have the largest negative effect on precision, with an ATE of $-0.185$. Miscellaneous libraries also decrease precision, with an ATE of $-0.106$. In contrast, Android and Small Libraries have effects close to zero.

For CryptoGuard, the effects are different. Android libraries have a small negative effect on precision, with an ATE of $-0.056$. However, Utilities, Small Libraries, and miscellaneous libraries show positive effects, with ATEs of $0.139$, $0.129$, and $0.079$, respectively. This suggests that, unlike CogniCrypt, including several categories of third-party library alerts in CryptoGuard does not reduce precision.

For Semgrep, the positive effect comes from Small Libraries, with an ATE of $0.077$. Android libraries also show a smaller positive effect, with an ATE of $0.050$. Utilities and miscellaneous libraries have effects close to zero, suggesting no strong precision change for those categories.

For CodeQL, miscellaneous libraries have the clearest negative effect, with an ATE of $-0.115$. Android libraries show a very small positive effect, with an ATE of $0.026$. Utilities and Small Libraries do not show strong evidence of a precision change.

**Finding 9 ($\mathcal{F}_9$) –** The causal effect of reporting third-party library alerts is heterogeneous across both library families and SASTs. Utilities substantially reduce precision for CogniCrypt but increase precision for CryptoGuard; miscellaneous libraries reduce precision for CogniCrypt and CodeQL; and Android and Small libraries exhibit small positive, small negative, or negligible effects depending on the SAST.

**3. Robustness of CAUSEC's causal estimates (EQ3):** We leverage the refutation tests discussed in Section 5.6 to evaluate the robustness of the causal estimates obtained for **EQ1** and **EQ2**. Detailed refutation results are provided in Table 3 and Table 5 in Appendix C. Here, we focus on the overall robustness trends and notable exceptions.

For $EQ1$, the estimates for CogniCrypt, Semgrep, and CodeQL are stable across the refutation tests, as seen in Table 3 in Appendix C. We interpret a refutation-test p-value below 0.05 as evidence that the refuted estimate differs significantly from the original estimate, indicating sensitivity to that particular perturbation. The random common cause refuter leaves the estimates unchanged, the placebo and dummy outcome refuters produce near-zero effects, and the data subset refuter produces effects close to the original estimates. For example, under the data subset refuter, the new effects are $-0.263$ for CogniCrypt, $0.073$ for Semgrep, and $-0.074$ for CodeQL, all with p-values $>= 0.05$.

CryptoGuard is the only tool for which the $EQ1$ PSM estimate is both inconsistent with the assumed direction of the effect and sensitive to the placebo refuter. We therefore treat the positive PSM estimate as a robustness-sensitive result rather than strong evidence that reporting third-party alerts improves precision. A likely explanation is CryptoGuard's alert composition. First, the raw precision gap between developer-written and third-party alerts is small ($0.513$ vs. $0.479$), making the adjusted estimate sensitive to perturbation. Moreover, CryptoGuard's third-party alerts are heterogeneous: Android alerts have low precision, while Utilities and Social Media alerts have much higher precision. Finally, CryptoGuard's rule behavior is highly polarized, with some high-volume rules producing mostly false positives and others producing mostly true positives. Together, these factors suggest that the estimated effect is sensitive to the composition of alerts included in the analysis, which may explain its instability under refutation.

To further assess the robustness of our **EQ1** estimates, we repeated the analysis using the logistic regression adjustment as an alternative estimator. We used the same causal graph, treatment, outcome, and confounders as in the primary PSM analysis. As shown in Table 4 in Appendix C, the alternative estimator confirms the direction of the effect for CogniCrypt, Semgrep, and CodeQL. For CryptoGuard, however, the direction changes from a small positive PSM estimate to a small negative logistic-regression estimate. Together with the placebo-refuter result discussed above, this suggests that the CryptoGuard estimate is sensitive to the choice of estimator and should be interpreted cautiously.

For **EQ2** as well, the refutation results are also largely stable, as seen in Table 5 in Appendix C Across the four tools and library-type contrasts, the random common cause refuter leaves the estimates unchanged, and the placebo and dummy outcome refuters generally produce near-zero effects with non-significant p-values of $>= 0.05$. The main exceptions occur under the data subset refuter: CodeQL's Android and miscellaneous contrast, and CryptoGuard's miscellaneous contrasts, show significant changes under data resampling. These cases suggest that some library-type estimates are sensitive to the sampled subset of the data and should be interpreted more cautiously. The remaining **EQ2** estimates remain stable under the refutation tests.

To further assess the robustness of our **EQ2** estimates, just as **EQ1**, we repeated the library-type analysis using logistic regression adjustment as an alternative estimator. As shown in Table 2 in Appendix C, the logistic-regression estimates generally agree with the PSM estimates: 13 out of 16 library-type contrasts have the same direction. The main direction changes occur for Android libraries in CogniCrypt and CodeQL, and miscellaneous libraries in CryptoGuard. Two of these cases have effects close to zero under at least one estimator, suggesting that the corresponding library-type effects should be interpreted cautiously rather than as strong contradictions. Overall, the alternative estimator supports our main conclusion that the causal effect of reporting third-party library alerts varies across both tools and library categories.

**Finding 10 ($\mathcal{F}_{10}$) –** CAUSEC's adjusted effect estimates are largely stable across the refutation tests, supporting the robustness of the main empirical findings under the specified causal model. However, robustness is not uniform: CryptoGuard's **EQ1** estimate is sensitive to the placebo refuter, and a small number of **EQ2** estimates, particularly for CodeQL and CryptoGuard, are sensitive to the data subset refuter. Therefore, we interpret the **overall causal trends as reliable**, while treating these specific estimates as robustness-sensitive cases.

## 8. Threats to Validity

**Manual analysis and labeling:** One researcher with three years of experience in crypto-API misuse and vulnerability analysis extracted assumptions from the literature and manually labeled all alerts in the evaluation dataset. While single-researcher extraction and labeling are common in large-scale studies in this area [43], [44], we cannot completely eliminate the possibility of human error or subjective judgment. To mitigate this risk, a second researcher with more than fifteen years of experience in vulnerability analysis reviewed the extracted assumptions, supervised the labeling process, and independently reviewed the causal modeling decisions, results, and findings.

**Internal validity:** Our causal estimates depend on the appropriateness of the structural causal models and backdoor adjustment sets specified in CAUSEC. Therefore, the reported effects should not be interpreted as assumption-free causal effects; rather, they are adjusted effect estimates whose validity depends on the specified DAG, measured confounders, and identification assumptions. If important confounders are omitted (*e.g.,* app domain, developer expertise, or unobserved code quality) or edges in the DAG are mis-specified, residual confounding may bias the estimated effects of treatments such as reporting third-party library alerts. We mitigate this risk by deriving the DAGs from domain knowledge, checking their structural plausibility through empirical association tests, inspecting robustness through multiple refutation tests, and comparing the main PSM estimates against logistic-regression adjustment.

**External validity:** We evaluate CAUSEC on crypto-API misuse detection in Android using 558 Android apps repre-

senting 11 popularity strata and four tools (Semgrep, CodeQL, CogniCrypt, and CryptoGuard). Although this covers diverse real-world apps and state-of-the-art detectors, the empirical results may not generalize to other SAST tools, platforms, or industrial codebases. In addition, our evaluation focuses on a single assumption and treatment family; other assumptions identified in our SLR may exhibit different causal effects. While the CAUSEC framework is designed to support a broader range of SAST assumptions, additional studies are needed to understand how its findings generalize across assumptions, tools, and domains.

## 9. Discussion and Takeaways

We now distill the results of our assumption study, the evaluation of CAUSEC, and the resulting 10 findings ($\mathcal{F}_1$–$\mathcal{F}_{10}$) into three discussion themes that conclude with takeaways for SAST designers and the broader security community.

### 9.1. Diverse assumptions to test, but there are limits to what can be tested

Our analysis of assumptions from just 20 SAST papers related to crypto-detectors led to 57 security assumptions. These assumptions were diverse in terms of their conditions. Particularly, we found several assumptions that put significant emphasis on the rules used by crypto detectors and the eventual outcome (*e.g.,* precision) ($\mathcal{F}_1$). We also found security assumptions regarding the impact of various static analysis techniques and optimizations (*e.g.,* slicing, adding context, heuristics to define scope) that were expected to improve both precision and runtime ($\mathcal{F}_3$), with some indicating deliberate design decisions sacrificing detection based on the assumed effects ($\mathcal{F}_4$). Finally, we also found assumptions that considered end-user outcomes, such as developer satisfaction or productivity ($\mathcal{F}_5$). This diverse range of assumptions provides a fertile ground for applying CAUSEC to evaluate their effects under explicit causal models. However, there is a caveat: *not all assumptions are equally amenable to causal evaluation in practice.*

To elaborate, considering the ground truth data we curated, assumptions regarding detector rules can often be evaluated with minimal additional effort, since the corresponding treatments can be realized directly on the alert dataset (*e.g.,* by including or suppressing alerts from a specific rule or ruleset). In contrast, assumptions related to static-analysis techniques or heuristics typically require method-level treatments that modify the SAST itself. The effort involved would depend on how configurable the SAST is and how tightly coupled the technique (e.g., slicing or exploration depth) is with the overall analysis. Finally, some assumptions involve outcomes that are difficult to measure in controlled experiments (*e.g.,* developer satisfaction), making them currently infeasible to evaluate causally.

**Takeaway 1**

The availability of diverse assumptions signifies both a problem and a research opportunity, particularly because many of these assumptions can be operationalized and evaluated systematically. However, the nature of their conditions and outcomes determines how readily they can be operationalized for causal evaluation, with challenges ranging from the complexity of modifying SASTs to obtaining field data.

### 9.2. Causal Inference is an effective tool

Through our evaluation with CAUSEC, we observe that causal inference is an effective tool for systematically testing security assumptions that are amenable to causal analysis ($\mathcal{F}_7$ – $\mathcal{F}_{10}$). The resultant causal effect is not only robust under refutation ($\mathcal{F}_{10}$), but, in some cases, reveal effects that are substantially different from those suggested by one-off, unadjusted, observations ($\mathcal{F}_7$). Thus, CAUSEC provides a valuable addition to the design and evaluation process of SASTs by replacing ad-hoc validation that does not account for confounding ($\mathcal{F}_2$) with a principled causal analysis that does. This is particularly true in evaluations where labeled alert data is already available and can be used to construct data-level treatments for assumptions involving rule inclusion, code inclusion, or similar reporting decisions.

**Takeaway 2**

Causal inference provides an effective alternative to ad-hoc validation of SAST assumptions and can be readily integrated into evaluation workflows when the necessary data and treatments can be constructed.

### 9.3. We cannot *inherit* valid assumptions

Our evaluation across four SASTs leads to a key lesson: tool design acts as an effect modifier ($\mathcal{F}_8$). Even when evaluating the same assumption on the same application dataset, different tools can exhibit substantially different estimates of the causal effect. In our case, the assumption that reporting third-party library alerts decreases precision holds for some tools but not for others.

This phenomenon changes how we learn from prior work, as it exposes a key lesson for security researchers and designers of SASTs: *even if an assumption holds for a prior tool, we cannot assume that it will hold for a new one*. Differences in rule design, implementation choices, and other tool-specific characteristics can substantially alter the estimated effect of the same treatment. Consequently, there is no guarantee that a broadly accepted assumption will transfer across SASTs.

**Takeaway 3**

As SASTs are *effect modifiers*, the causal effect cannot be assumed to be transferable across tools, even for the same assumption and evaluation dataset. Thus, when evaluating design choices, the underlying assumptions should be tested in the context of the specific SAST under development rather than inherited from prior tools.

## References


[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[2] "Coverity SAST — Static Application Security Testing by Black Duck," https://www.blackduck.com/static-analysis-tools-sast/coverity.html, [Accessed 02-02-2026].

[3] Z. D. Wadhams, C. Izurieta, and A. M. Reinhold, "Barriers to using static application security testing (sast) tools: A literature review," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ser. ASEW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 161–166. [Online]. Available: https://doi.org/10.1145/3691621.3694947

[4] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering Flaws in Security-Focused Static Analysis Tools for Android Using Systematic Mutation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, 2018, pp. 1263–1280.

[5] A. S. Ami, N. Cooper, K. Kafle, K. Moran, D. Poshyvanyk, and A. Nadkarni, "Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 614–631.

[6] P. Mandal, A. S. Ami, V. Olaiya, S. H. Razmjo, and A. Nadkarni, ""Belt and suspenders" or "just red tape"?: Investigating Early Artifacts and User Perceptions of IoT App Security Certification," in *Proceedings of the 2024 USENIX Security Symposium (USENIX)*, Aug. 2024.

[7] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: Supporting Developers in Using Cryptography," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 931–936.

[8] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS '19*. London, United Kingdom: ACM Press, 2019, pp. 2455–2472.

[9] "Fix: Scan apk classes with android. substring package name," https://github.com/CryptoGuardOSS/cryptoguard/pull/8, [Last Accessed 02-02-2026].

[10] J. Pearl and D. Mackenzie, *The Book of Why: The New Science of Cause and Effect*, 1st ed. USA: Basic Books, Inc., 2018.

[11] Anonymous, "Causec: Unboxing the causal drivers of static vulnerability analysis performance - online appendix," https://anonymous.4open.science/r/CauSec/, 2026, last accessed on June 11, 2026.

[12] J. Pearl, M. Glymour, and N. P. Jewell, *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.

[13] J. Pearl, *Causality: Models, Reasoning and Inference*, 2nd ed. USA: Cambridge University Press, 2009.

[14] D. Rodriguez-Cardenas, A. Garryyeva, D. N. Palacio, A. Mastropaolo, and D. Poshyvanyk, "Rethinking software empirical studies with structural causal models," 2026. [Online]. Available: https://arxiv.org/abs/2605.28482

[15] D. N. Palacio, A. Velasco, N. Cooper, A. Rodriguez, K. Moran, and D. Poshyvanyk, "Toward a theory of causation for interpreting neural code models," *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1215–1243, 2024.

[16] J. Pearl, "Causal inference in statistics: An overview," *Statistics Surveys*, vol. 3, pp. 96–146, 2009. [Online]. Available: https://projecteuclid.org/journals/statistics-surveys/volume-3/issue-none/Causal-inference-in-statistics-An-overview/10.1214/09-SS057.full

[17] L. G. Neuberg, "Causality: Models, reasoning, and inference," *Econometric Theory*, vol. 19, no. 4, pp. 675–685, 2003, review of Pearl's causal inference framework.

[18] D. Rodriguez-Cardenas, D. N. Palacio, D. Khati, H. Burke, and D. Poshyvanyk, "Benchmarking causal study to interpret large language models for source code," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 329–334.

[19] A. Velasco, D. Rodriguez-Cardenas, D. Khati, D. N. Palacio, L. R. Alif, and D. Poshyvanyk, "A Causal Perspective on Measuring, Explaining and Mitigating Smells in LLM-Generated Code," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. Rio de Janeiro, Brazil: IEEE/ACM, Apr. 2026. [Online]. Available: http://arxiv.org/abs/2511.15817

[20] H. Yang, A. Velasco, S. Fang, B. Xu, and D. Poshyvanyk, "Understanding Privacy Risks in Code Models Through Training Dynamics: A Causal Approach," Dec. 2025, arXiv:2512.07814 [cs]. [Online]. Available: http://arxiv.org/abs/2512.07814

[21] H. Yang, A. Velasco, T. Le-Cong, M. N. Haque, B. Xu, and D. Poshyvanyk, "How Do Semantically Equivalent Code Transformations Impact Membership Inference on LLMs for Code?" in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. Rio de Janeiro, Brazil: IEEE/ACM, Apr. 2026. [Online]. Available: http://arxiv.org/abs/2512.15468

[22] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[23] X. O. Fengguo Wei, Sankardas Roy and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014.

[24] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[25] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale," in *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)*, Jun. 2012.

[26] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, May 2015, pp. 426–436.

[27] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android SSL (in)Security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61.

[28] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14*, 2014.

[29] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2021.

[30] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 73–84.

[31] "Find security bugs," https://find-sec-bugs.github.io/, [Accessed 02-02-2026].

[32] "Code quality, security & static analysis tool with sonarqube," https://www.sonarsource.com/products/sonarqube/, [Accessed 02-02-2026].

[33] S. Lab, "Github security lab," https://securitylab.github.com/, [Accessed 06-02-2026].

[34] "Shiftleftsecurity/sast-scan," https://github.com/ShiftLeftSecurity/sast-scan, [Accessed 06-02-2026].

[35] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android," *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 6, Jun. 2014.

[36] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "Towards Formal Analysis of the Permission-based Security Model for Android," in *Proceedings of the 5th International Conference on Wireless and Mobile Communications*, 2009.

[37] B. Livshits, D. Vardoulakis, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, and A. Mø ller, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015.

[38] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A comprehensive benchmark on java cryptographic API misuses," in *2019 IEEE Cybersecurity Development (SecDev)*, Sep. 2019, pp. 49–61.

[39] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "Taintbench: Automatic real-world malware benchmarking of android taint analyses," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–41, 2022.

[40] M. Frantz, Y. Xiao, T. S. Pias, N. Meng, and D. Yao, "Methods and benchmark for detecting cryptographic api misuses in python," *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1118–1129, 2024.

[41] H. Wang and Y. Guo, "Understanding third-party libraries in mobile app analysis," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 515–516.

[42] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[43] Y. Chen, Y. Liu, K. L. Wu, D. V. Le, and S. Y. Chau, "Towards precise reporting of cryptographic misuses." in *NDSS*, 2024.

[44] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, "" false negative-that one is going to kill you": Understanding industry perspectives of static analysis based security testing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3979–3997.

[45] W. Li, S. Jia, L. Liu, F. Zheng, Y. Ma, and J. Lin, "Cryptogo: Automatic detection of go cryptographic api misuses," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 318–331.

[46] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, and N. Meng, "Example-based vulnerability detection and repair in java code," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 190–201.

[47] G. Bennett, T. Hall, E. Winter, and S. Counsell, "Semgrep*: Improving the limited performance of static application security testing (sast) tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 614–623.

[48] "Semgrep," https://semgrep.dev/, [Accessed 06-06-2026].

[49] L. Singleton, R. Zhao, H. Siy, and M. Song, "Firebugs: Finding and repairing cryptography api misuses in mobile applications," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 1194–1201.

[50] A. Sharma and E. Kiciman, "Dowhy: An end-to-end library for causal inference," 2020. [Online]. Available: https://arxiv.org/abs/2011.04216

[51] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 468–471.

[52] "Codeql," https://codeql.github.com/, [Accessed 06-06-2026].

[53] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and Its Security Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 356–367.

# Appendix A. Additional Background on Causal Inference

**Estimands:** Causal effects are quantified through five primary estimands, each addressing a distinct question, *i.e.,* the First, the *Average Treatment Effect (ATE)* computes the expected difference in outcomes between treatment and control groups across the entire population, answering "on average, what is the effect for everyone?". Second, the *Average Treatment Effect on the Treated (ATT)* quantifies the causal effect specifically for units that actually received treatment, answering "what is the effect for those who were treated?". Third, the *Average Treatment Effect on the Control (ATC)* indicates the causal effect for untreated units, estimating what would happen if they were treated, answering "what would be the effect for those who remained untreated?". Fourth, the *Conditional Average Treatment Effect (CATE)*: estimates the treatment effect estimated within subgroups or conditional on specific covariates, capturing heterogeneous treatment effects when effectiveness varies across populations. Finally, the *Individual Treatment Effect (ITE)* estimates the treatment effect for a single unit, representing the difference in potential outcomes for that individual. ITE is fundamentally unobservable in any individual case but theoretically informs personalized interventions. While ATE, ATT, and CATE are empirically estimable from data, ITE remains latent but serves as a conceptual foundation for heterogeneous treatment effect estimation.

| Tool | Setting | # of Alerts | Prec. |
|---|---|---|---|
| Semgrep | **Overall (all alerts)** | **14590** | **0.6743** |
| | Developer-written only (is_third_party=0) | 1760 | 0.5494 |
| | Third-party only (is_third_party=1) | 12830 | 0.6914 |
| | Balanced A (downsampled) - Combined | 3520 | 0.6227 |
| | Balanced A - Third-party (downsampled) | 1760 | 0.6960 |
| | Balanced B (upsampled) - Combined | 25660 | 0.6208 |
| | Balanced B - Developer-written (upsampled) | 12830 | 0.5501 |
| CodeQL | **Overall (all alerts)** | **21501** | **0.8887** |
| | Developer-written only (is_third_party=0) | 554 | 0.9513 |
| | Third-party only (is_third_party=1) | 20947 | 0.8870 |
| | Balanced A (downsampled) - Combined | 1108 | 0.9233 |
| | Balanced A - Third-party (downsampled) | 554 | 0.8953 |
| | Balanced B (upsampled) - Combined | 41894 | 0.9193 |
| | Balanced B - Developer-written (upsampled) | 20947 | 0.9515 |
| CogniCrypt | **Overall (all alerts)** | **5675** | **0.4846** |
| | Developer-written only (is_third_party=0) | 498 | 0.7008 |
| | Third-party only (is_third_party=1) | 5177 | 0.4638 |
| | Balanced A (downsampled) - Combined | 996 | 0.5914 |
| | Balanced A - Third-party (downsampled) | 498 | 0.4819 |
| | Balanced B (upsampled) - Combined | 10354 | 0.5863 |
| | Balanced B - Developer-written (upsampled) | 5177 | 0.7089 |
| CryptoGuard | **Overall (all alerts)** | **15272** | **0.4817** |
| | Developer-written only (is_third_party=0) | 1318 | 0.5129 |
| | Third-party only (is_third_party=1) | 13954 | 0.4788 |
| | Balanced A (downsampled) - Combined | 2636 | 0.4981 |
| | Balanced A - Third-party (downsampled) | 1318 | 0.4833 |
| | Balanced B (upsampled) - Combined | 27908 | 0.4962 |
| | Balanced B - Developer-written (upsampled) | 13954 | 0.5135 |

TABLE 1: Precision Baselines Before Causal Adjustment. Balanced A = Downsample alerts from third-party lib to match alerts from developer-written count; Balanced B = Upsample alerts from developer-written to match alerts from third-party count

# Appendix B. Analysis of Assumptions

## B.1. SLR Search Query

We used the following search query for finding our initial list of papers in identified sources for our systematic literature review:
*("static analysis" OR "SAST" OR "static application security testing") AND ("taint analysis" OR "taint tracking" OR "data leak detection" OR "crypto API misuse" OR "cryptographic API misuse" OR "API misuse" OR "vulnerability detection")*

# Appendix C. CAUSEC Evaluation Baselines, Estimates, and Refutation Results

| Tool | Library Type | PSM ATE | Logistic ATE | Same Direction? |
|---|---|---|---|---|
| Semgrep | Android | 0.050 | 0.057 | Yes |
| | Small Libraries | 0.077 | 0.072 | Yes |
| | Utilities | 0.011 | 0.045 | Yes |
| | Miscellaneous | 0.001 | 0.058 | Yes |
| CodeQL | Android | 0.026 | −0.004 | No / mixed |
| | Small Libraries | 0.016 | 0.002 | Yes |
| | Utilities | −0.006 | −0.023 | Yes |
| | Miscellaneous | −0.115 | −0.079 | Yes |
| CogniCrypt | Android | −0.002 | 0.029 | No / mixed |
| | Small Libraries | 0.008 | 0.109 | Yes |
| | Utilities | −0.185 | −0.204 | Yes |
| | Miscellaneous | −0.106 | −0.056 | Yes |
| CryptoGuard | Android | −0.056 | −0.077 | Yes |
| | Small Libraries | 0.129 | 0.126 | Yes |
| | Utilities | 0.139 | 0.118 | Yes |
| | Miscellaneous | 0.079 | −0.016 | No / mixed |

TABLE 2: Comparison of PSM and logistic regression adjustment estimates for $EQ2$.

| Tool | ATE | 95% CI | Refuter | New Effect | Estimated Effect | p-value |
|---|---|---|---|---|---|---|
| Semgrep | 0.109 | [0.107, 0.202] | Random common cause | 0.109 | 0.109 | 1.000 |
| | | | Placebo treatment | −0.016 | 0.109 | 0.560 |
| | | | Data subset | 0.073 | 0.109 | 0.100 |
| | | | Dummy outcome | −0.011 | 0.000 | 0.840 |
| CodeQL | −0.057 | [−0.072, −0.017] | Random common cause | −0.057 | −0.057 | 1.000 |
| | | | Placebo treatment | 0.008 | −0.057 | 0.720 |
| | | | Data subset | −0.074 | −0.057 | 0.190 |
| | | | Dummy outcome | −0.008 | 0.000 | 0.880 |
| CogniCrypt | −0.283 | [−0.381, −0.225] | Random common cause | −0.283 | −0.283 | 1.000 |
| | | | Placebo treatment | −0.004 | −0.283 | 0.910 |
| | | | Data subset | −0.263 | −0.283 | 0.650 |
| | | | Dummy outcome | 0.002 | 0.000 | 1.000 |
| CryptoGuard | 0.043 | [0.018, 0.150] | Random common cause | 0.043 | 0.043 | 1.000 |
| | | | Placebo treatment | 0.080 | 0.043 | 0.020 |
| | | | Data subset | 0.001 | 0.043 | 0.150 |
| | | | Dummy outcome | −0.004 | 0.000 | 1.000 |

TABLE 3: Causal Estimates and Refutation Results for $EQ1$.

# Appendix D. CAUSEC Implementation

We implemented the CAUSEC using the *do-why* library [50], which supports causal analysis from a well-structured dataset. We first encoded the causal graph from Figure 2 as a DAG in DoWhy's graph specification language. The treatment $T$ is the inclusion of alerts from third-party libraries (*3rd-party-lib)*, the outcome $Y$ is the alert correctness (verdict), and the confounders $Z$ include *app_popularity* and

| Tool | PSM ATE | Logistic ATE | Same Direction? |
|---|---|---|---|
| Semgrep | 0.109 | 0.122 | Yes |
| CodeQL | −0.057 | −0.064 | Yes |
| CogniCrypt | −0.283 | −0.223 | Yes |
| CryptoGuard | 0.043 | −0.030 | No / mixed |

TABLE 4: Comparison of PSM and logistic regression adjustment estimates for $EQ1$.

*apk_size*. For estimation, we use *DoWhy's* propensity-score matching estimator configured for a binary treatment, in this case *PSM*. We then validated the graph with correlation tests and applied *DoWhy's* refuters (*i.e.,* random common cause, placebo, and data subset) to probe robustness.

# Appendix E. Causal Effect Estimators

## E.1. Propensity Score Matching (PSM)

Propensity Score Matching estimates causal effects by first computing, for each observation, the probability of receiving the treatment given the observed adjustment variables. This probability is called the propensity score. Treated observations are then compared with control observations that have similar propensity scores. The intuition is that, among observations with similar propensity scores, the treated and control groups have similar observable contexts, making the comparison more balanced. The causal effect is then estimated by comparing the outcomes of matched treated and control observations and averaging these differences across the matched sample.

## E.2. Logistic Regression Adjustment

Logistic regression adjustment estimates the causal effect by directly modeling the outcome as a function of the treatment and the adjustment variables. Because our outcome is binary, true positive or false positive, logistic regression is a natural model for estimating the probability that an alert is a true positive.

After fitting the model, the treatment effect is estimated by predicting each observation twice: once as if it received the treatment, and once as if it did not, while keeping the adjustment variables fixed. The difference between these two predicted probabilities gives the estimated effect for that observation. Averaging these differences over all observations gives the average treatment effect.

Unlike PSM, which compares matched treated and control observations, logistic regression adjustment uses an outcome model to estimate how the treatment changes the probability of a true positive after accounting for the adjustment variables. We use it as an alternative estimator to check whether the direction and approximate magnitude of the estimated effect are consistent with the PSM results.

| Tool | Library | ATE | Random Common Cause | Placebo Treatment Refuter | Data Subset Refuter | Dummy Outcome Refuter |
|---|---|---|---|---|---|---|
| Semgrep | Android | 0.050 | 0.050 (1.000) | −0.003 (0.840) | 0.036 (0.520) | −0.005 (0.990) |
| | Misc. | 0.001 | 0.001 (1.000) | −0.002 (0.920) | 0.033 (0.090) | −0.005 (0.970) |
| | Utilities | 0.011 | 0.011 (1.000) | −0.006 (0.840) | 0.002 (0.720) | 0.004 (0.940) |
| | Small Lib. | 0.077 | 0.077 (1.000) | −0.006 (0.980) | 0.041 (0.080) | 0.004 (1.000) |
| CodeQL | Android | 0.026 | 0.026 (1.000) | 0.002 (0.875) | −0.005 (0.030) | −0.011 (0.890) |
| | Misc. | −0.115 | −0.115 (1.000) | 0.004 (0.915) | −0.059 (0.000) | −0.011 (0.950) |
| | Utilities | −0.006 | −0.006 (1.000) | 0.001 (0.915) | −0.022 (0.260) | 0.002 (0.970) |
| | Small Lib. | 0.016 | 0.016 (1.000) | −0.000 (1.000) | 0.008 (0.490) | −0.006 (0.990) |
| CogniCrypt | Misc. | −0.106 | −0.106 (1.000) | 0.003 (0.940) | −0.062 (0.300) | −0.012 (0.910) |
| | Utilities | −0.185 | −0.185 (1.000) | 0.001 (0.935) | −0.175 (0.780) | −0.008 (0.960) |
| | Android | −0.002 | −0.002 (1.000) | 0.012 (0.790) | −0.002 (0.960) | −0.008 (0.950) |
| | Small Lib. | 0.008 | 0.008 (1.000) | 0.010 (0.730) | 0.056 (0.190) | 0.007 (0.940) |
| CryptoGuard | Misc. | 0.079 | 0.079 (1.000) | 0.002 (0.970) | 0.020 (0.010) | −0.007 (0.980) |
| | Android | −0.056 | −0.056 (1.000) | 0.003 (0.960) | −0.036 (0.360) | 0.013 (0.900) |
| | Utilities | 0.139 | 0.139 (1.000) | 0.001 (0.930) | 0.096 (0.060) | −0.001 (0.980) |
| | Small Lib. | 0.129 | 0.129 (1.000) | 0.004 (0.925) | 0.115 (0.550) | 0.018 (0.900) |

TABLE 5: Causal Estimates and Refutation Results for $EQ2$. Each refuter cell reports a new effect with a p-value in parentheses.