Squeezing Juicy Variant Bugs Out of Modern Browsers

Han Zheng
EPFL

Flavio Toffalini
Ruhr-Universität Bochum

Qiang Liu
EPFL

Mathias Payer
EPFL

Abstract

Complex software systems, like web browsers, integrate multiple tightly-coupled components. While code reviews and fuzzing enhance code quality, eliminating all bugs remains infeasible due to large-scale projects, unpredictable cross-context interactions, and complex cross-domain dependencies. This dire situation calls for an approach that scales to this unprecedented complexity.

Inspired by informal variant analysis developed by the hacker community, we create GRAPE, *a structured approach* that supports analysts in writing rules to detect bugs. By focusing on code patterns, GRAPE scales effectively to large-scale code projects. Moreover, our novel variant bug model enables analysis of cross-context interactions and exploitability verification using existing bug reports, eliminating the need for cross-domain dependencies. GRAPE represents the *first systematic approach to variant analysis*, introducing principles for variant pattern development.

We implement a prototype of GRAPE, which scans the entire Chromium code base in only 12 minutes. GRAPE discovered 24 new bugs, with four assigned CVEs and 17,500 USD in rewards from Chrome's Vulnerability Rewards Program. These discoveries impact modern web browser and security-critical complex software like OpenSSL. Beyond browsers, GRAPE uncovered three logic bugs in VSCode and Azure Data Studio, one of which received a CVE from Microsoft.

1 Introduction

Unlike standalone programs, complex software systems, such as web browsers, combine independent components to provide rich functionalities. By exploiting subtle interactions between components, attackers can build a vulnerable state in the victim system and trigger unpredictable errors. Given these sophisticated techniques, even more advanced mitigations cannot guarantee complete protection and leave an exploitable attack surface, as demonstrated by the long trail of discovered vulnerabilities [4,5,44,52,60,72]. As a result, *finding such vulnerabilities before release* is essential to reduce the risk of exploitation.

Researchers have proposed numerous approaches for bug detection, including static analysis [7, 8], model checking [26,43], automated testing (fuzzing) [3,29,30,32,42,51, 75, 78, 81, 86, 92], and symbolic execution [9]. Fuzzing, in particular, has become the industry standard for bug discovery. Complementing fuzzing, rigorous code reviews [15] and extensive unit testing [16] are common industry practices. However, due to the software's inherent complexity, finding security bugs is challenging as modelling all possible system states and exploring the whole input space remains elusive.

To complement existing techniques, We explore a novel strategy centered on the concept of *variant bugs*, which are bugs that share the same root cause as a previously fixed bug but remain in the codebase [36]. The technique of identifying such bugs is referred to as *variant analysis*. Two pragmatic observations drive our approach. First, the hacker community successfully employs ad-hoc manual variant analysis to identify vulnerabilities in the real world [36, 65, 94]. Evidently, variant analysis has the potential to detect bugs, and we argue this approach can apply to prevention as well. Second, our systematization of 342 Chromium bug reports (see Section 2.1) reveals three key findings about variant bugs. They occur frequently (approximately 10% [14]), persist in codebases 2.26 times longer than regular bugs (averaging three years [14]), and directly threaten system integrity (forming the basis for around 40% of exploits [67]). Therefore, our study suggests that complex projects, exemplified by Chromium, suffer from variant bugs. However, despite the promise of variant analysis and sustained investigation by the research community [76, 80], this methodology has not yet been integrated integrated in the development process. The reason roots in the considerable manual effort required to analyze patterns and create regular expressions along with the risk of high volume of false negatives as showed in current implementations (see Section 7). While there is large potential and dire need, these limitations must be addressed to make variant analysis effective for complex code bases.

From all the works discussed above, we identify three major sources of complexity that translate to corresponding challenges in safeguarding complex software systems, focusing specifically on web browsers.

Challenge 1: Code Size. The scale of a web browser prevents comprehensive analysis across diverse code paths, rendering traditional formal methods and symbolic execution impractical. Even with extensive unit testing, fuzzing reaches its limits. For instance, fuzzing covers only 29% of Chromium's code base [25, 61].

Challenge 2: Cross-Context Interactions. Asynchronous events, such as garbage collection, complicate accurate modeling of object lifecycles [73] and introduce an opaque data-flow that hampers the development of secure code. For example, since Chromium UI objects can be destroyed at any time, failing to verify object liveness can result in Use-After-Free (UAF) [20, 22, 24].

Challenge 3: Cross-Domain Dependencies. Web browsers combine several programming languages (*e.g.,* JavaScript and C++) and diverse modules (*e.g.,* PDF and HTML parsers) to support complex functionalities (*e.g.,* UI interactions and PDF rendering). Despite these interactions being regular and expected, bridging data and control flows across these diverse contexts requires advanced techniques to handle cross-domain dependencies.

To address approach for these challenges, we propose GRAPE, the *first structured variant analysis*. By focusing on code patterns, GRAPE scales to handle arbitrary Code Size (C1). Additionally, it incorporates a standardized bug variant model to address Cross-Context Interactions (C2), and introduces an exploitability verification method based on variant categories, eliminating the need for complex Cross-Domain Dependencies analysis (C3). Our novel variant model removes the burden of writing complex patterns and reduces analyst effort to identifying just a few simple code snippets in an existing bug report (Section 3). More specifically, a security analyst needs to identify only *four* key code locations from the report, namely: context, violation, assumption, and abuse. Figure 1 showcases a practical example taken from Chromium, where context (1), assumption (2), violation (3), abuse (4) indicate the four elements of our model, respectively. Based on this simple code annotation, GRAPE seeks other variant bugs without needing to rewrite possibly incomplete expressions.

We subsequently illustrate how this standardized model detects real-world vulnerabilities in large-scale software such as Chromium, AzureDataStudio, and VSCode. Specifically, we showcase the model's effectiveness by identifying six distinct bug patterns in Chromium, spanning diverse components across all process privilege layers. When compared to grepping with regular expressions, our model reduces false positives by 86.3% (from 168 to 23), highlighting its effectiveness in improving accuracy.

To evaluate GRAPE's effectiveness and scalability, we build a prototype by extending SemGrep [69] and apply it to the latest Chromium with 46M LoC. GRAPE successfully identified 24 confirmed vulnerabilities, receiving 17,500 USD rewards through Chrome's Vulnerability Rewards Program. Furthermore, our findings impact Firefox and Safari due to their integration of Skia and WebRTC [33, 34, 55]. Our approach detects the bugs in 12 minutes, with each full scan completing in under 9 minutes. Notably, GRAPE uncovered bugs that survived on average for 5.2 years despite extensive testing. Finally, we demonstrate GRAPE's extensibility by discovering a bug in OpenSSL [59] and three non-memory corruption vulnerabilities in VSCode [56], all of which have been fixed and acknowledged by the vendor.

```
// [1] Context
void CallbackLayerAnimationObserver::SetActive() {
  weak_this = GetWeakPtr();
  // [3] Violation
  CheckAllSequencesStarted();
  // [2] Assumption
  if (!weak_this)  { return; }
  // [4] Abuse
  CheckAllSequencesCompleted();
}
```

Figure 1: A didactic example of GRAPE code patterns.

In summary, our contributions are:

- We conduct a large-scale bug analysis on modern browsers to understand and systematize variant bugs.
- We propose a variant model based on observed properties to establish GRAPE, the *first variant analysis workflow*.
- We open source our prototype at https://github.com/HexHive/Grape, which discovers 24 confirmed memory and logical vulnerabilities, impacting Chromium, OpenSSL, and VSCode, yielding bug bounties totalling 17,500 USD.

2 Bug Variant in Real-World Projects

This section examines the characteristics of variant bugs and presents the current practice for variant bug findings. We focus on Chromium because it is a complex, widely used open-source project with a well-maintained issue tracker and detailed bug reports [14]. Our study classifies variant bugs, identifies their key properties, and assesses their impact on Chromium's security. Based on the observed variant properties, we introduce the current industry and academia techniques to hunt for variant bugs.

Data Collection. Chromium's rapid mitigation development mitigates prior vulnerabilities or renders them non-exploitable [13, 88]. As a result, older vulnerabilities may no longer be relevant. To reflect the current security landscape, we analyze vulnerabilities reported from October 1, 2023, to October 1, 2024. We focus on bounty-awarded reports, as they indicate security impact, and exclude internal

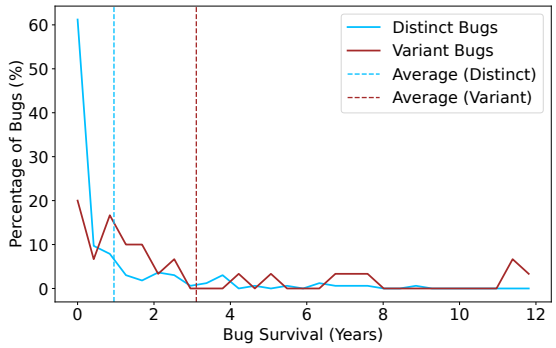

Figure 2: Average Bug Survival years for different types of bugs, measured from bug introduction to bug reporting.

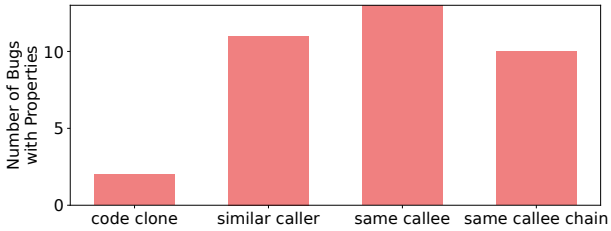

Figure 3: Number of variant bugs across the four properties. One variant bug may have multiple properties.

reports to avoid duplicates and ClusterFuzz cases that only internal developers can access [39]. After manual deduplication, we collect 342 unique reports.

Bug Categories. We categorize the 342 bug reports into three major types: memory corruption (252 reports), logic bugs (71 reports), and mitigation bypasses (19 reports). We observe that 73.7% are memory corruption bugs, posing most significant threat to web browser [18]. Considering the large fraction of bugs and strong security implications, our study primarily focus on the memory corruptions vulnerabilities [66].

Data Processing Methodology. As bug reports are written in natural language, we manually analyze them. Specifically, we first extract all available fields from the issue tracker, then manually conduct variant analysis based on the defined metrics (Section 2.2).

2.1 Bug Variant Analysis

Based on the 252 memory corruption reports from Chromium, we observe two distinct groups:

Distinct Bugs refer to bugs that are initially discovered independently, either later identified as related to a fixed bug or determined to have no direct relation to any known bug.

Variant Bugs are identified through root cause analysis, exhibiting the same underlying bug pattern as a previously reported vulnerability. These bugs are subsequently discovered based on prior knowledge of the shared bug model.

After manual analysis, we observe the followings:

Variant Bugs are better suited for exploitation. Figure 2 illustrates our key observations regarding the survival times of different bug categories, *i.e.,* variant bugs are significantly more difficult to detect, with an average discovery time of over three years. In contrast, distinct bugs are identified more quickly, with an average survival time of 347 days. This indicates that variant bugs require approximately 2.26 times longer to be detected compared to distinct bugs. According to Google Project Zero, well-designed PoCs for high-severity vulnerabilities are typically long-lived, as their discovery often requires considerable effort. If a bug is fixed immediately after its introduction, it is less likely to be exploited effectively. Consequently, bugs with longer survival periods are generally more stable for exploitation [87]. Therefore, we conclude that variant bugs are generally more severe than others.

Variant Bugs remain overlooked. Despite the significant impact of variant bugs, the number of such bugs detected remains relatively small. Among 252 memory corruption reports, a considerable amount of bugs (32) are variants of a known report. According to a Google study [67], over 40% of discovered 0-day vulnerabilities are variants of known bugs, highlighting that variant analysis is a crucial but often overlooked aspect of vulnerability detection. So far, existing research missed this pattern.

Variant Bugs affect different components. Variant bugs are widespread in Chromium, with 32 reports across 12 components. Eight affect UI-related components, potentially corrupting the browser process, while seven impact privileged processes including GPU and network. Renderer components are also vulnerable. These bugs contribute to various memory corruption issues, including spatial and temporal flaws. Effective variant analysis must comprehend diverse components.

2.2 Bug Variant Properties

After inspecting 32 variant bugs, we identify four key properties based on their similarity to previously known vulnerabilities, as illustrated in Figure 3. Note that a single bug may exhibit multiple properties. Furthermore, these properties are not exhaustive but rather reflect the state of Chromium issues at the time of writing. In Section 3, we describe how to leverage these properties for practical bug pattern development. Finally, all data from this study is included in our publically available at `https://github.com/HexHive/Grape`.

Code Clone. Some memory corruption bugs originate from a single copy-pasted line of code. For example, in *b/339877158* and *b/339788215*, the *CSSTokenizer().TokenizeToEOF()* line was copy-pasted, leading to a Use-After-Free vulnerability when receiving similarly crafted input. In our study, we found two cases of this pattern. Moreover, Code Clone is easily

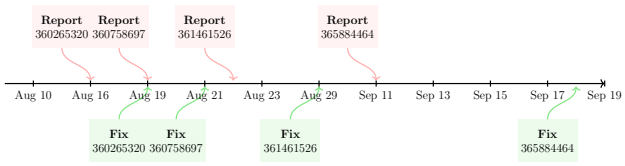

Figure 4: Timeline for incomplete fix CVE-2024-8193 (Addition Integer Overflow). All reports were in 2024.

identifiable by string-based tools or regular expressions.

Similar Caller. Chromium exposes similar functionalities through the class hierarchy or if-else code branches. While these code segments are not direct copies, they share similar structures with minor variations (*e.g.,* changing only the variable name). These patterns can be identified by analyzing the caller function or class names, which tend to remain consistent. For example, the fix for *b/355256380* addresses the *MaglevGraphBuilder*'s method *InferHasInPrototypeChain*, whereas *b/367818758* shows that the *JSNativeContextSpecialization* also contains a similar method vulnerable to type confusion. The most effective approach to handling these variants is to search for sibling names related to the affected class. Eleven bug reports exhibit the *Similar Caller* property, often concealed within sibling classes or functions of the original bug report.

Same Callee. Another common scenario for bug variants arises when the root cause involves the same function or variable. However, in this case the vulnerability is not introduced by a single line of code. For instance, the fix for *b/40053095* swaps *Resolve(...)* and *Init(...)* to address a UAF issue, thus altering the code pattern. Moerover, *b/339588211* finds that the code following *Init(...)* involves a UAF due to potential object destruction caused by *Resolve(...)*. Detecting these bugs without variant analysis is challenging since searching for the variable or function name alone leads to over-approximation, while code pattern matching may result in under-approximation. *Same Callee* bugs constitute 40.6% of the variant reports.

Same Callee Chain. The root cause of certain bugs consists in multiple variables or functions used in different code locations. For these cases, an ad-hoc fix is often insufficient, as similar errors can be triggered with slightly dfferent call chains. For example, in *b/41484151*, the class *BaseRenderingContext2DAutoRestoreSkCanvas* indirectly calls *GetOrCreatePaintCanvas*, which destroys the canvas, and results in a UAF. In *b/41493290*, a similar issue occurs with the same root cause: *GetOrCreatePaintCanvas* destroys the canvas through the function *WillDraw*. This illustrates that bugs with similar root cause can manifest in different locations. This property comprises nine reports that bypass the ad-hoc fixes.

2.3 Finding Variant Bugs Manually

Industry bug hunters rely on their expertise for variant analysis [41,58,65,94], but manual analysis is often incomplete and unstructured. Despite their effort, some variant bugs remain. In Chromium, both internal developers and external security researchers primarily focus on *code clone* and *similar caller* bugs, often missing other variants. Figure 4 illustrates this issue: after a bug report on August 16, 2024 (*b/360265320*), a series of related *similar caller* bugs was found. Despite an exhaustive search over the next month, all detected variants followed the *similar caller* pattern, failing to uncover issues in the *same callee chain*. Four months later, GRAPE identified a previously undetected *same callee chain* variant (Figure 14) using its bug variant model. Similarly, Section 7.4 highlights a case where manual audits only found *code clone* variants, whereas GRAPE uncovered a *same callee chain* vulnerability. These results demonstrate GRAPE 's ability to detect a broader range of variant bugs beyond manual methods.

2.4 Finding Variant Bugs using the State of the Art

Despite significant advancements in vulnerability detection techniques, existing approaches are not designed to find variant bugs. To address this gap, we analyze current vulnerability-finding methods from three key perspectives: code size, cross-context interactions, and cross-domain dependencies, and we compare their strengths and limitations. Moreover, we examine speed, false positives (FP), false negatives (FN), and usability to discuss trade-offs between different approaches. Table 1 summarizes our analysis.

Model checking translates code into a state transition graph and verifies if the code meets given specifications, proving program correctness [26,43]. This approach inherently avoids FPs and FNs. However, the code complexity of modern web browsers poses an unsurmountable challenge, as model checking is prone to state explosion. To date, there have been no successful attempts to scale model checking to code bases as complex as Chromium.

Symbolic execution represents program variables symbolically and converts code conditions into symbolic constraints [2,9,48], which are processed by constraint solvers (*e.g.,* SMT solvers [27]). While symbolic execution can theoretically achieve full code coverage and handle cross-context interactions and cross-domain dependencies, it similarly suffers from state explosion. The tens of millions of lines of code in modern browsers result in path constraints that are too complex to solve, limiting scalability.

Coverage-guided fuzzing (CGF) generates thousands of inputs per second to maximize code coverage by mutating a minimized seed corpus [6,32,51,86]. CGFs incrementally and stochastically explores the near-infinite state space, resulting in a manageable FP rate, as crashes typically indicate vul-

Table 1: Comparison of major bug-finding techniques for open-source programs. ^: Web browser fuzzing produces intentional crashes [73], necessitating manual triaging. *: Tested on an i7-13700 CPU. +: including the required Chromium compilation

Technique	Genre	Code Size	Code Coverage	Handle Cross-Context	Handle Cross-Domain	False Positive	False Negative	Testing Chromium*
Model Checking [26, 43]	Static	1K LoC [53]	High	Yes	Yes	No	No	Timeout
Symbolic Execution [9, 27]	Static	10K LoC [9]	High	Yes	Yes	No	No	Timeout
Concolic Execution [70, 85]	Dynamic	50K LoC [70]	Medium	Partitial	Partitial	No	Medium	Timeout
Coverage-Guided Fuzzing [32]	Dynamic	500K LoC [79]	Low	Partitial	Partitial	Low^	Medium	> 1 week
Graph-Based Analyzer [37]	Static	>10M LoC [61]	High	No	No	Medium	High	> 6 h+
GRAPE (Section 3.2)	Static	>10M LoC [61]	High	Yes	Yes	Low	Medium	< 5 min

nerabilities (except intentional crashes [73]). However, CGF often hits a coverage plateau, leaving code areas unexplored—particularly in complex software like web browsers [25], where only 29% of Chromium code is covered by fuzzing. Additionally, cross-context interactions and cross-domain dependencies remain challenging, as not all code paths are tested.

Concolic Execution combines concrete and symbolic execution, starting with a concrete path and abstracting variables to explore nearby paths [10, 64, 70, 85]. While this approach reduces state explosion, it still relies on symbolic variables and SMT solvers. As program size grows, handling complex path constraints becomes challenging, limiting scalability. Additionally, the reduced path coverage leads to FNs and failure at addressing cross-context interactions and cross-domain dependencies.

A **graph-based analyzer** compiles source code into Abstract Syntax Trees (ASTs) [37, 82, 83] and uses manually written queries to identify vulnerabilities. These analyzers scale well to arbitrary C/C++ programs as they avoid state explosion through limited query states. However, their manually written queries are tailored to specific bug scenarios, and are thus prone to both FPs and FNs. Additionally, as graph-based analyzers operate on metadata instructions, they struggle with cross-context interactions and cross-domain dependencies [7, 8, 82, 83]. The reliance on compilation also limits regression testing, as code changes lead to a full recompilation. For complex software like Chromium, this is infeasible as it requires over 43 core hours for a single end-to-end scan [38].

3 GRAPE

This section first outlines three key challenges in detecting variant bugs in complex software systems (Section 3.1). Then, we introduce GRAPE (Section 3.2), an extensible, implementation-independent workflow that is compatible with any static analyzer.[1]

[1]The term GRAPE refers to a phonetic play on words that hints at the informal approach of using the Unix tool GREP to search for bug patterns.

3.1 Challenges

C1: Code Size (Where to look for bugs). Large-scale software projects are inherently complex and cannot be exhaustively analyzed; for example, in Chromium, only 29% of its codebase is covered by tests [25]. *This discrepancy highlights the difficulty of thoroughly testing complex systems, making it a major challenge to identify which areas to prioritize.*

C2: Cross-Context Interactions (How to detect bugs). The management of object lifetimes in large software systems is complex and difficult to track [73]. Unlike explicit memory management, which allows for predictable patterns (*e.g.,* allocation, destruction, and reuse in Use-After-Free bugs), the object release process in complex systems is often triggered by user actions and does not involve explicit *free* functions. *This irregular data-flow hinders the usage of traditional bug models, scaling bug detection framework to modern software therefore becomes the second challenge.*

C3: Cross-Domain Dependencies (How to verify exploitability). Bugs are considered security vulnerabilities only if an attacker can exploit them from a known entry point. For instance, an integer overflow caused by missing parameter validation might not be exploitable if security checks are properly placed. This challenge is exacerbated in complex software systems where data and control flows span multiple components, leading to a state explosion that traditional models struggle to manage. *Even with a potential bug report, determining its exploitability remains a significant hurdle.*

To tackle these challenges, we leverage variant analysis by introducing a standardized bug variant model and verifying the bugs exploitability through variant-specific properties.

Addressing C1. We narrow the target scope by leveraging variant analysis. Specifically, by using a known bug as input, we reduce the search space from the entire codebase to regions directly related to the identified vulnerability.

Addressing C2. We introduce a formalized bug variant model (Section 4) that abstracts the implicit operations into explicit expressions. The model compresses expert knowledge into machine-readable patterns, automating the bug detection.

Addressing C3. We present a method for verifying exploitability, leveraging the variant categories defined in Section 2.2. This step ensures that identified vulnerabilities are

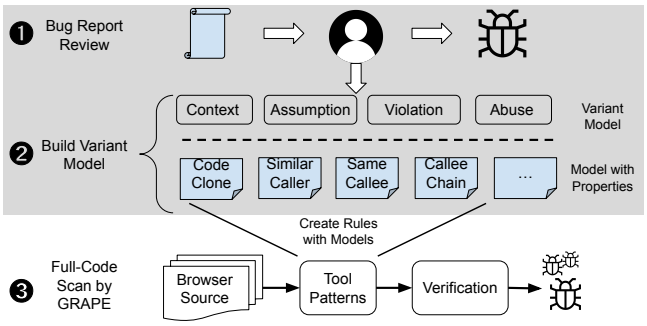

Figure 5: GRAPE's design. ■ indicates one-time efforts.

assessed in the context of their security impact, refining the overall detection framework.

3.2 GRAPE Design

To address these limitations, we propose GRAPE: an analyzer-agnostic variant-analysis approach that minimizes reliance on heavyweight source code knowledge. GRAPE avoids state explosion and seamlessly scale to large codebases like Chromium. Figure 5 describes our design. We leverage a knowledge-driven model that translates human expertise into string-based abstract bug descriptions (Section 4), further removing the need for AST dependencies and allowing a lightweight implementation.

GRAPE incorporates new bug patterns based on three key steps. (❶) **Bug Report Review**: A developer reviews existing bug reports and identifies the assumption, violation, and abuse. (❷) **Building the Variant Model**: Using these insights, the developer creates properties-specific patterns for variant detection by leveraging the general variant model. (❸) **Code Scan**: GRAPE applies these patterns across the codebase, detecting similar vulnerabilities and verifying their exploitability.

How does GRAPE simplify variant analysis? The underlying message of GRAPE is to write sound bug patterns *only once*. Given an existing pattern, analysts need only to identify a few specific code strings in an existing patch. In more detail, steps ❶ and ❷ are one-time efforts, requiring no further maintenance. Crucially, developers implement the first two steps while producing a bug patch, thus not introducing additional efforts. In our experience, steps ❶ and ❷ requires approximately five minutes to a first year PhD student, while we expect an expert would reduce this time (more details in Section 4.5). Finally, step ❸ automatically executes the patterns over the whole code base. The lightweight nature of this step makes it suitable for integration into CI/CD pipelines for improving code quality.

We implement GRAPE by extending SemGrep [69], eliminating the heavy recompilation cost of graph-based analyzer while maintaining our detection efficiency. In Section 4.4, we provide concrete instructions and illustrate an concrete query writing process.

4 Variant Models with Different Properties

This section details GRAPE's bug variant model and how it overcomes the limitations of current practices, which are tailored to specific targets [45, 58]. Specifically, GRAPE's model provides a systematic and generic approach, thus addressing cross-context interactions. In Section 4.1, we present a highly abstract model applicable to all variant bug categories. We then refine this model in Section 4.2 to detect variants adhering to the characteristics outlined in Section 2.2. Our approach is designed to provide flexibility while remaining extensible to accommodate new properties.

4.1 Variant Bug Model

Our variant bug model consists of four key elements:
Context: A context is the class, function or code block that contain the given calls. For instance, if a bug is located in function *ClassA::VulnFunc*, its context is *ClassA::VulnFunc*. The developer leverages this information to search for *Similar Caller* bug variants.
Assumption: An assumption is an abstract representation of the expected runtime state at a specific code location. For example, object *X* should always be alive while function *Y* is executing. Developers infer assumptions from bug reports and use them to create two rules that represent *violation* and *abuse*, respectively.
Violation: A violation is a condition that contradicts a valid *assumption*. For instance, an attacker could trigger a violation by accessing a freed object *X* via function *Y*, thus violating the developer's assumption. Each violation is encoded into a rule.
Abuse: Violations alone are only harmful when followed by an *abuse*, turning the pair into a vulnerability. Abuse occurs when a code segment operates on a violated condition, creating a vulnerability. For example, accessing a member of a freed object, such as *X.a()*, constitutes an abuse. As with violations, abuses are also encoded into rules.

4.2 Variant Bug Models with Properties

Our property findings in Section 2.2 highlight the potential for refining the variant model. We demonstrate how the standardized model in Section 4.1 can be adjusted based on observed variant properties. Due to the generality of our model (Section 4.1), further refinements can be integrated with existing solutions.
Bug Model For Code Clone. *Code clone* variants arise when vulnerable code is directly copied and pasted. As a result, the string pattern for identifying such variants is a line of code that encapsulates all the necessary conditions for triggering the vulnerability, *i.e.,* both *violation* and *abuse*. For

```
void CallerFuncLinux() {
  Violation1();
  Abuse1();
}
```

(a) Prior Bug

```
void CallerFuncWin() {
  Violation2();
  Abuse2();
}
```

(b) New Bug

Figure 6: Variant Model for *Similar Caller*s.

instance, in a Use-After-Free (UAF) vulnerability, the string pattern should include the complete sequence of memory allocation, deallocation, and reuse. Given this structure, detecting instances of `Violation(); Abuse();` is generally sufficient for real-world bug detection.

Bug Model For Similar Caller. For similar caller variants, the developer need to find a function or class that is similar to the current buggy function. In complex software systems, it is common practice to reuse implementations across sibling classes or functions. Security researchers focusing on these variants should examine sibling functions or class names associated with the caller of a known bug. As illustrated in Figure 6, if a vulnerability is present in *CallerFuncLinux*, a platform-specific counterpart such as *CallerFuncWin* is likely to exhibit the same issue.

```
void AnyFuncA() {
  Violation();
  Abuse1();
}
```

(a) Prior Bug

```
void AnyFuncB() {
  Violation();
  Abuse2();
}
```

(b) New Bug

Figure 7: Variant Model for *Same Callee*s.

Bug Model For Same Callee Variant. Bugs involving the same callee tend to be complex and cannot always be summarized in a few lines of code. Therefore, the *violation* function should serve as an approximation based on the most critical condition necessary for triggering the bug. As demonstrated in Figure 7, for a UAF vulnerability, the function responsible for freeing memory can be used as the *Violation* to identify potential variants, even if different functions handle the *Abuse* (*i.e.,* memory reuse).

Bug Model For Same Callee Chain Variant. When a callee is invoked in multiple locations, an attacker may reach a vulnerable function from different code paths. In such cases, the model must capture runtime checks occurring before the function call and identify where those checks are missing. Figure 8 illustrates two scenarios: one in which the absence of inter-function validation results in a variant bug, and another where missing intra-function validation leads to a similar vulnerability. These cases underscore how differences in validation mechanisms can contribute to the emergence of new bug variants. This variant aligns with the *Same Callee* model. While the *Same Callee* approach restricts its analysis to *Abuse*

```
void Function1() {
  Violation1();
+  if (!Validate())
+    return ;
  Abuse1();
}
```

(a) Prior Bug

```
// [1] intra-func
Violation1();
if (!Validate())
  return ;
...
Violation1();
Abuse1();

// [2] inter-func
Function1();
Abuse1();
```

(b) New Bug

Figure 8: Variant Model for *Same Callee Chain*s.

instances that occur in proximity to the current *Violation*, the *Same Callee Chain* expands this scope by considering *Abuse* across a broader sequence of related calls.

4.3 Writing Queries using Variant Models

The variant models with properties in Section 4.2 only illustrate the concept for variant bug detection, and cannot be directly convert to queries for static tools. We present concrete examples on how these concepts inspire actual query writing.

Query for *Code Clone*. For a *code clone* variant to be exploitable, the copied code must retain the necessary bug conditions, including both *violation* and *abuse*. Therefore, the detection query must check for the presence of both conditions. For example, queries for *b/339458177* should use *CSSTokenizer(*).TokenizeToEOF()*, encompassing both *violation* and *abuse*.

Query for *Similar Caller*. Variants with the *similar caller* property occur when a known vulnerable function is called by multiple sibling classes that implement similar functionality. These classes may propagate the same vulnerability. Take *b/360265320* for instance, as *MeshOp* is proven to be vulnerable, developer can leverage their knowledge that *FillRectOp* may suffer from the same issue, and searching for *FillRectOp*'s *onCombineIfPossible* implementation to add approximate validations.

Query for *Same Callee*. The *same Callee* property simplifies the detection of vulnerabilities by focusing first on *violation* conditions, which serve as an approximation for full bug conditions. For instance, in *b/40064490*, two *SetOptionSelection()* calls constitute full bug conditions. While the first call is a *violation* that frees the object, the second call triggers an *abuse*, *i.e.,* it dereferences a freed object. Instead of searching for repeating *SetOptionSelection()* calls, we can look for the single *SetOptionSelection()* first, then validate if an *abuse* follows.

Query for *Same Callee Chain*. Variants with the *same callee chain* property extend the analysis of vulnerabilities across different contexts by considering inter-function relationships. Compared to *same callee* property, which focuses on vulnera-

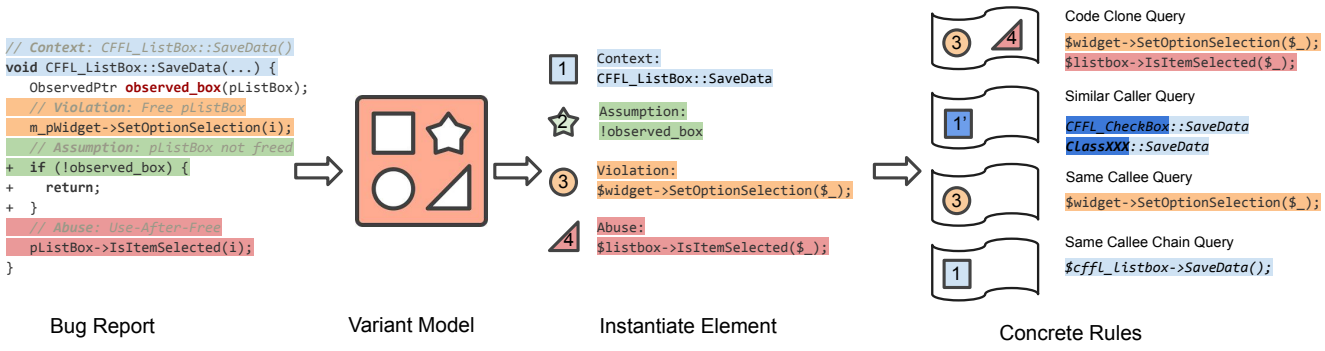

Figure 9: Main steps to incorporate a bug report into GRAPE.

bilities within a single function, this property tracks vulnerabilities that may span multiple functions. In *b/360265320*, the overflow happens during the *onCombineIfPossible* calculation, which is later used for space allocation inside *onPrepareDraws*. Thus, the developer should also track the value propagation across the *Same Callee Chain*, *i.e.,* searching for the *fIndexCount* calculation chain.

4.4 Query Writing Example

As illustrated in Figure 9, this section details the process by which a developer can create concrete static analyzer queries from a bug report, corresponding to steps ❶ and ❷ in the GRAPE overall design (Figure 5). The core of this process involves using a variant model to systematically deconstruct a bug. This model functions as a template, where a developer abstracts key components (*i.e.,* the **Context**, **Assumption**, **Violation**, and **Abuse**) from the bug-related code. By combining these instantiated code elements, concrete query rules can be formulated for a static analyzer.

The process begins with the construction of an abstract **variant model**. To exemplify this, we consider cross-language invalidation bugs. This class of vulnerability arises when a user-defined callback, often in a scripting language like JavaScript, deallocates a native object (*e.g.,* a C++ object representing part of a PDF document), leading to a subsequent use-after-free. For such bugs, the variant model is defined by the following four components:

We consider the **Context** as the function handling the specific document features (*e.g.,* PDF XFA forms) that can trigger script execution. Located in this **Context**, the developer's **Assumption** is that all native objects remain valid throughout the execution of a member function. However, the **Assumption** will be **violated** when attacker-controlled callback embedded in the document executes and prematurely destroys a C++ object. Finally, the **Abuse** occur when the freed objected is reused, resulting in a Use-After-Free.

Following this abstract model, the developer instantiates each element by mapping it to specific code artifacts from the bug report. In the given example: The **Context** is identified as the `CFFL_ListBox::SaveData` method, which is the function encompassing the vulnerable code. The developer's **Assumption** is inferred from the post-patch code. The added check `if (!observed_box)` reveals the original code's implicit assumption that the `pListBox` object would not be deallocated during the function's execution. The call to `m_pWidget->SetOptionSelection(i)` is identified as the **Violation**, as this is the operation that may trigger the script callback and deallocate the object, thereby violating the assumption. Finally, the subsequent call to `pListBox->IsItemSelected(i)` in the original code represents the **Abuse**, where the potentially freed `pListBox` object is used, leading to the use-after-free.

With these instantiated elements, a developer can compose various concrete query rules. These rules are generally based on code patterns and can be adapted to any static analysis tool that supports syntactic or string-based matching. As shown in Figure 9, different combinations of the instantiated elements yield distinct query types:

- A **Code Clone Query** combines the **Violation** and **Abuse** patterns (`$widget->SetOptionSelection($_); $listbox->IsItemSelected($_);`) to find exact duplicates of the bug.
- A **Similar Caller Query** generalizes the **Context**. Instead of only searching `CFFL_ListBox::SaveData`, it expands the search to semantically similar methods, like `CFFL_CheckBox` or `CFFL_ComboBox`.
- A **Same Callee Query** focuses exclusively on the **Violation** element (`$widget->SetOptionSelection($_);`), aiming to find all locations where the potential violation occurs.
- A **Same Callee Chain Query** searches for callers of the function containing the violation. In this example, since `CFFL_ListBox::SaveData()` return when `pListBox` is destroyed, any code that calls this method also destroy `pListBox`, results in an *violation*. Subsequent use of the `pListBox` object may introduce a new vulnerability. This query, `$cffl_listbox->SaveData();`, identifies such call sites to uncover more complex variants.

Table 2: Required manually identified Variables (V), Functions (F) and Classes (C). If the number is 0, the variant property does not apply to the bug pattern.

Bug Patterns	Bug Property				
	Code Clone	Similar Caller	Same Callee	Same Callee Chain	All
Cross-Language Invalidation	2F	2C	1F	1F	2C3F
Unexpected UI Destruction	2F	5C	1F	1F	5C3F
Inconsistent Variable Lifetime	1C1F	0C	1F	3C	3C2F
Multiply Integer Overflow	2F	7C	1F	0F	7C2F
Add Integer Overflow	1F	6C	1V	3V	6C1F3V
Improper Access Control	2F	0C	1F	1F	2F

4.5 Manual Efforts

Overall, GRAPE suggests a promising direction for reducing manual effort. The only human intervention necessary is selecting a few relevant variables. To quantitatively assess this manual effort, we selected the most complex bug report associated with each bug pattern and measured the number of variables, function names, and class names the developer must identify when following the instructions in Section 4.4. Table 2 summarizes our findings. Across all evaluated cases, the maximum manual effort involved identifying no more than seven class names and two function names for any single bug. These results demonstrate the efficiency and practicality of the GRAPE workflow, even when applied to complex, real-world vulnerabilities such as those found in the Chromium.

5 Implementation

Building on the variant model and properties illustrated in Section 4, we implement a GRAPE prototype using SemGrep [69] and scan the Chromium codebase to assess its effectiveness. GRAPE systematically translates natural language bug descriptions into concrete detection queries. While GRAPE is currently implemented upon SemGrep, its pattern design is independent of any specific analyzer, allowing for potential extensions to other static analyzers, such as CodeQL [37]. Since GRAPE primarily relies on string-based matching, SemGrep remains a lightweight and efficient choice.

6 Bug Patterns and Our Discoveries

Following the workflow in Section 4.4, we conduct variant analysis on real-world software systems. Specifically, we use Chromium and VSCode as case studies, and derive six patterns (see Table 3). Furthermore, we discuss the OpenSSL pattern in Appendix A.4.

Pattern 1: Cross-Language Invalidation. Modern web browsers support scripting languages, such as JavaScript (JS), embedded in web pages and PDF documents. The interaction between JS and C++ code happens through a binding layer that handles cross-language communications. By manipulating JS callbacks in PDF or HTML documents, attackers can control C++ APIs and free C++ objects. These vulnerabilities represent a strong primitive that adversaries employ in real-world scenarios [49]. The discovery of this pattern resulted in the assignment of two CVEs.

Pattern 2: Unexpected UI Destruction. Since the User Interface (UI) resides in the browser process, outside the sandbox, it serves as an alternative, and often overlooked, attack surface. Consequently, adversaries can exploit UI memory safety violations to gain the highest privilege. Despite significant efforts, UI memory corruption remains a substantial threat to web browser security [20, 22, 24]. By scanning existing bug reports, we find a universal bug variant: Unexpected UI Destruction. This vulnerability relies on an intrinsic browser policy: when necessary, the web browser deconstructs some UI objects if not visible for user interaction. Therefore, an incorrect check of UI objects' lifetime may lead to a UAF. GRAPE discover seven bugs based on this pattern.

Pattern 3: Inconsistent Variable Lifetime. Since Chromium can consume significant memory due to its numerous components, developers tend to use object with a short lifecycle. For instance, they often declare temporary variables on the stack instead of the heap. When stack variables hold computation results, the caller receives a copy rather than a reference of the stack object, ensuring safety in most of the cases. However, in certain edge cases, references to destroyed stack objects may mistakenly persist, resulting in exploitable UAF vulnerabilities. This type of error is prevalent throughout the Chromium project, often safeguarded by complex conditions (*e.g.,* GPU shutdown [19], extension installation [23], timing windows [21]) that hinder fuzzers from detection. Additionally, the allocation of stack variables, along with the creation and dereferencing of references, occurs across different contexts. Therefore, the system needs to reach sophisticated internal state to trigger the bug. As a result, current static analysis tools are unable to address these vulnerabilities. In our study, we identify the Inconsistent Variable Lifetime bug pattern and detect one high-severity UAF instance.

Pattern 4: Multiply Arithmetic Overflow. Web browsers integrate third-party libraries for file parsing and decoding. Unfortunately, many of these libraries were developed decades ago and often lack robust security practices. When allocating heaps buffers, these libraries frequently fail to validate

Table 3: Overview of our six bug variants. Bug variant type stands for the discovered variant bug type.

Program	Vulnerability Type		Privilege	Bug Variant Pattern	Bug Variant Type			
					Code Clone	Similar Caller	Same Callee	Same Callee Chain
Chromium	Temporal	Use-After-Free	Render	Cross-Language Invalidation	✗	✗	✓	✓
			Browser	Unexpected UI Destruction	✗	✓	✓	✓
		Use-After-Return	Render	Inconsistent Variable Lifetime	✓	✗	✗	✓
	Spartial	Heap-Buffer-Overflow	Render	Multiply Integer Overflow	✓	✓	✓	✗
			GPU	Add Integer Overflow	✗	✗	✓	✓
VSCode	Logical	Information Leak	-	Improper Access Control	✓	✗	✓	✗

whether the integer operation exceed the maximum limits, resulting in integer overflow. Although Chromium's security model classifies integer overflow as undefined behavior rather than a security vulnerability [74], these bugs can cause security issues if the overflowed values control pointer arithmetic calculations. In these cases, the allocation may produce an unexpectedly small memory chunk, resulting in heap out-of-bounds accesses, and finally leading to a security abuse. Based on this pattern, GRAPE detects nine vulnerabilities without the need to test the complex internal states of browsers.

Pattern 5: Add Arithmetic Overflow. The increasing complexity of web applications demands enhanced graphics computation capabilities, prompting browsers to integrate GPU support for accelerated rendering and to integrate graphics libraries, such as Skia [71]. During shape rendering, loops often perform calculations over indices and vertices. However, insufficient validation can result in integer overflows. When the buffer size calculation relies on overflowed values, the attacker may induce heap buffer overflow through small memory buffers. Regarding this pattern, GRAPE identified a High-Severity vulnerability that allows a sandbox escape in Chrome GPU component.

Pattern 6: Improper Access Control. During desktop application development, JavaScript and TypeScript are widely used. While these memory safe languages mitigates traditional memory corruption vulnerabilities, they introduces new security risks, *e.g.,* Improper Access Control (CWE-284). These vulnerabilities arise when the applications store confidential messages in a temporary directory, allowing the adversary, another user in the system, to inspect the secret or inject the code, achieving information leakage or even privilege elevation. Leveraging this variant pattern, GRAPE successfully identified three logic flaws in VSCode and Azure Data Studio, Microsoft has acknowledged two of them.

7 Real-World Bug Hunting

In this section, we evaluate whether GRAPE can find real-world vulnerabilities in complex software systems, Section 7.1 summarizes our discoveries. We first discuss the performance and required manual effort. Then, we investigate the False Positive rate for GRAPE in Section 7.2. Finally, we study six bug patterns from Section 7.3 to 7.5. Due to page limitation, we discuss Pattern 1 in Appendix A.1 , Pattern 4 in Appendix A.2 and the OpenSSL bug in Appendix A.4.

Performance. GRAPE completes an end-to-end pattern scan in up to 9 minutes, while completing all patterns takes only 12 minutes. Performance was measured over five runs on a Ubuntu 22.04 desktop with an i7-13700 CPU and 64GB RAM.

Manual Effort. Among the 49 GRAPE reports, we manually excluded 23 false positives (FPs) within two hours, demonstrating a reasonable effort requirement. Rule creation is a one-time process that largely overlaps with bug fixing [17].

Table 4: New bugs reported by GRAPE. '*' indicates the report include multiple bugs. 'b/0123' means https://crbug.com/0123.

ID	Pattern	Component	Type	Severity	Status	Bounty
1	P1	Plugin >PDF	Vuln	Medium	CVE-2024-5847	1000
2	P1	Plugin >PDF	Vuln	Medium	CVE-2024-5846	1000
3	P2	Headless	Bug	Low	b/347737878	0
4	P2	UI >Shell	Bug	Medium	b/348688192	0
5	P2	UI >Autofill	Vuln	High	CVE-2024-7968	1000
6	P2	UI >Aura	Bug	Medium	b/351796118	0
7	P2	Unclassified	Vuln	Medium	b/351843813	500
8	P2	Views >Desktop	Vuln	Low	b/363985581	0
9	P2	Unclassified	Vuln	High	b/363985598	0
10	P3	Blink >CSS	Vuln	High	b/365802556	11000
11*	P4	Blink >WebRTC	Vuln	High	b/371686447	0
12*	P4	Blink >WebRTC	Vuln	Medium	b/371615496	0
13	P5	Skia	Vuln	High	CVE-2025-0436	3000
14	P6	VSCode	Vuln	-	-	0
15	P6	AzureDataStudio	Vuln	Low	VULN-152863	0
16	P6	AzureDataStudio	Vuln	High	CVE-2025-32726	0
	In total, we reported 24 bugs in 16 reports and received 17,500 USD bounty					

7.1 New Vulnerabilities Found by GRAPE

Table 4 lists all 24 bugs identified by GRAPE. In total, GRAPE uncovered 21 vulnerabilities, with five classified as S1 (High-Severity), and was awarded a total of 17,500 USD. Five reports are categorized as bugs, this does not necessarily imply they are unrelated to security, as some security reviewers

Table 5: Bug detection with/without *abuse* filtering. We evaluate on Chromium 126.0.6465.2. "w/ abuse" stands for full GRAPE. *: GRAPE (w/ abuse) detect a new bug introduced in 130.0.6710.0, while w/o abuse failed.

Patterns	Violations	Without Filtering Output	Without Filtering FP rate	With Filtering Output	With Filtering FP rate	TP
Pattern 1	12	23	86.96%	5	40.00%	3
Pattern 2	68	150	95.33%	27	74.07%	7
Pattern 3*	1	8	50.00%	4	0.00%	4
Pattern 4	2	11	0.00%	11	0.00%	11
Pattern 5	1	2	50.00%	2	50.00%	1
Pattern 6	1	7	57.14%	7	57.14%	4
Overall	85	201	85.07%	56	46.43%	30

downgrade vulnerability reports without a PoC to bug. For instance, in reports 4 and 6, we provided the same analysis as in reports 5 and 7, which were initially classified as vulnerabilities by the Chromium security team. However, upon being forwarded to the ChromiumOS team, these were downgraded to bugs due to the lack of a concrete PoC. Additionally, we submitted 16 reports, as some reports (marked with *) contain multiple unique vulnerabilities with similar root causes.

7.2 False Positive and False Negative Rate

We evaluate the False Positive (FP) rate of GRAPE by running it on Chromium 126.0.6465.2 and VSCode 1.99.0, with and without the use of *Abuse* rules for FP pruning. Table 5 presents the results. We also discuss and approximate the False Negative (FN) rate of GRAPE. Overall, our variant model successfully eliminates 72.1% of FPs, lowering the FP rate from 85.1% to 46.4%, highlighting the importance a formalized variant model for more accurate detection. The full list of false positives and root causes is available at Appendix A.5.

Pattern 1 and Pattern 2. We automatize the *violation* detection by using `WeakPtr` validation (see Appendix A.3). By extracting call sites from functions exhibiting *violation* behavior, we initially identify 23 and 150 potential violations, respectively. With the the *abuse* model, the GRAPE prototype reduces the number of targets to 5 and 27, effectively eliminating a large portion of FPs.

Pattern 3. Despite relying on a manually defined *violation*, the FP rate remains high at 50%. Here, the introduction of the *abuse* model successfully eliminates all FPs. Additionally, in a later version of Chromium, a newly introduced vulnerability was detected by the GRAPE prototype, whereas a simple scan for *violation* alone failed to identify it. This highlights the importance of incorporating the *abuse* model for detection.

Pattern 4, Pattern 5 and Pattern 6. The *violation* conditions are manually defined, and the *abuse* model does not filter additional FPs. This is because the manually crafted *violation* already encapsulates sufficient knowledge to exclude potential FPs, reducing the need for additional filtering.

False Negative Rate. As it is inherently impossible to know the complete set of vulnerabilities in a system as large as Chrome, we estimate the false-negative rate by evaluating whether GRAPE can recover all known variants when initialized from an existing bug report. To the best of our knowledge, GRAPE successfully identifies all known variant cases reported in the chrome issuetracker [14].

7.3 Unexpected UI Destruction Variants

GRAPE identified seven unexpected UI destruction variants by leveraging the pattern 2 from Section 6, covering properties *Same Callee* and *Similar Caller*.

```
// crbug.com/995321, incomplete fix
driver_->GeneratedPasswordAccepted(...); // free
+ if (weak_this) { /* ... */ }

// new bug with same callee
driver_->GeneratedPasswordAccepted(...); // free
Show(...); // UAF
```

Figure 10: Incomplete fix of *b/995321*. GRAPE finds a High-Severity UAF and awarded 1K USD.

New Vulnerability with Same Callee. Figure 10 showcases a UI destruction bug found by GRAPE. In *b/995321*, developers noted that *GeneratedPasswordAccepted()* permits certain user gestures to destroy *this*. However, the function *EditPasswordClicked()* does not validate *this* after *GeneratedPasswordAccepted()*, resulting in an UAF. This bug was introduced in July 2023 [12] and was detected in June 2024. Despite a year of internal fuzzing, code audits, and static analysis, GRAPE revealed the vulnerability in just five minutes, highlighting GRAPE's effectiveness.

```
// crbug.com/335602634, only patch Linux
void xxTreeHostLinux::OnBoundsChanged(...) {
  xxTreeHostPlatform::OnBoundsChanged(change);
+ if (weak_this.get()) { ... }
}

// new bug with similar caller on Lacros
void xxTreeHostLacros::OnBoundsChanged(...) {
  xxTreeHostPlatform::OnBoundsChanged(change);
  // missing validation
}
```

Figure 11: Incomplete fix of *b/335602634*. GRAPE finds an unvalidated violation, which was confirmed and fixed.

New Vulnerability with Similar Caller. Figure 11 illustrates another bug where the fix only was applied to Linux implementation and does not propagate to Lacros (an experimental OS developed by Google). The implementation of *OnBoundsChanged* is exactly the same both in Linux and

Lacros, so does the unexpected UI destruction bugs. Our bug model easily catches the missed validation in 10 minutes, while this bug remained in the codebase for over 2 years.

7.4 Inconsistent Variable Lifetime Variants

We take the *CSSParserToken* series as an example to illustrate how GRAPE identifies inconsistency variant lifetime variants on exhaustively tested code.

Prior Attempts To Find Code Clone Vulnerabilities. Figure 13 illustrates the efforts of both developers and external researchers in detecting *Code Clone* variants. On May 8, 2024, developers identified that the call to *CSSTokenizer().TokenizerToEOF()* returns a pointer to a destroyed buffer, ultimately resulting in a use-after-free vulnerability. Over the following two weeks, numerous external security researchers and internal developers actively searched for similar variants. However, their approach was constrained to *Code Clone* variants. Five months after these initial bug fixes, GRAPE successfully identified a new *Same Callee Chain* variant by employing a more comprehensive bug model.

```
CSSParserToken GetAttrSubstitutionValue(...) {
  CSSParserTokenStream stream(attribute_value);
  // 'token' holds a reference to a stack object
  CSSParserToken token = \
  stream.ConsumeIncludingWhitespaceRaw();
  return token;
}

svalue = GetAttrSubstitutionValue(...);
// svalue point to a destroyed object
svalue->Serialize(serialized_svalue); // UAF
```

Figure 12: High-Severity UAF in Chromium Blink. Found by GRAPE and awarded 11K USD.

New Vulnerability with Same Callee Chain. Figure 12 showcases the new finding by GRAPE, which is a variant of CVE-2024-7000. GRAPE successfully identified a High-Severity UAR vulnerability where `CSSParserToken` stores a reference by a destroyed stack variable. Since `CSSParserToken` shares the *same callee chain* as `CSSTokenizer`, we infer that its input is equally under attacker control. Based on this assumption, we successfully constructed a PoC.

7.5 Addition Overflow Variants

GRAPE identifies a *similar callee chain* vulnerability by modeling the *violation* chain in GPU library.

Prior Attempts To Find Similar Caller Vulnerabilities. Figure 4 presents prior attempts to discover add-based overflows. Since the initial bug report *b/360265320* on August 16, bug hunters and developers have conducted extensive code audits on similar callers, specifically sibling classes of *MeshOp* that implement *onCombineIfPossible*. Although these audits

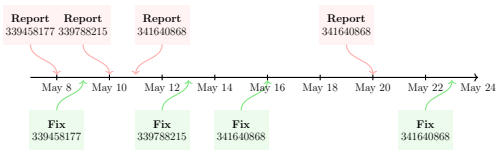

Figure 13: Timeline for incomplete fix CVE-2024-7000 (Inconsistency Variable Lifetime). All reports were in 2024.

```
// CVE-2024-8636: missing validation on mult
for (int i = 0; i < numRegions; i++) {
  numRects = safeMath.add(numRects, complexity);
}
// integer overflow in multiplication
int vertexCount = verticesPerRepete * numRects;
fVertices = target->makeVertexSpace(vertexCount);

// new finding: missing validation on addition
for (int i = 0; i < instanceCount; i++) {
  // integer overflow in addition
  quadCount += gather_lines_and_quads(...);
}
// vertexCount calculation is protected
// quadCount is used to calculate vertexCount
fVertices = target->makeVertexSpace(vertexCount);
```

Figure 14: High-Severity Integer Overflow in Chromium Skia, found by GRAPE and awarded 3K USD.

led to the discovery of four high-severity vulnerabilities, they missed the *same callee chain* variants.

New Vulnerabilities with Same Callee Chain. Figure 14 shows a new integer overflow identified by GRAPE, that similar to CVE-2024-8636. Both vulnerabilities arise from the accumulation of elements within a loop, followed by a multiplication operation to compute the allocation size (*vertexCount*). However, the two bugs lack distinct validations. In CVE-2024-8636, the developer employs a safe addition API to prevent overflow but fails to validate the multiplication that follows. Conversely, the new bug only safeguards the multiplication and forgets the addition operation. While both vulnerabilities occur within the *same callee chain*, different validations are missing. Notably, even with a developer-constructed PoC, reproducing the issue required weeks of execution. This complexity underscores the limitations of dynamic testing and highlights the GRAPE's effectiveness in identifying those overlooked bugs.

7.6 Inproper Access Control Variants

We extended GRAPE support to non memory corruption vulnerabilities. Specifically, we studied the two security-critical targets: AzureDataStudio [57] and VSCode [56]. Following the same workflow, we discover two logical vulnerabilities in AzureDataStudio and one in VSCode, one of which received a CVE from Microsoft. Figure 15 illustrates two examples.

```
// CVE-2025-20570, in VSCode
const zdotdir = path.join(os.tmpdir(), `xxx-zsh`);
envMixin['ZDOTDIR'] = zdotdir;

// Finding 1: Code Clone in AzureDataStudio
const zdotdir = path.join(os.tmpdir(), `xxx-zsh`);
envMixin['ZDOTDIR'] = zdotdir;

// Finding 2: Same Callee in VSCode
const certPath = path.join(os.tmpdir(), 'cert.pfx');
// ...
fs.writeFileSync(certPath, pfxCertificate);
// private key leak
cp.execSync(`openssl ... "${certPath}" ...`);
fs.rmSync(certPath, { force: true });
```

Figure 15: Incomplete fix of CVE-2025-20570. GRAPE finds a *Code Clone* and a *Same Callee*.

CVE-2025-20570 is an information leak vulnerability that allows a user within the same group to execute code under the victim user's role. Specifically, when a victim user (belonging to groupA) runs shell integration, the `zdotdir` is created in a directory where all users in groupA have full permissions. As a result, an attacker in groupA can inject code into the victim's .zshrc file, leading to arbitrary code execution with the victim's privileges and causing an information leak.

New Vulnerabilities with Cross-Repository Code Clone. The above vulnerability is presented in VSCode, and we notice that AzureDataStudio reuses the same code snippet. Specifically, as a fork of VSCode, AzureDataStudio provides the same shell support by porting the code of VSCode. Thus, by searching for the combination of *violation* and *Abuse*, *i.e.,* `zdotdir = path.join(os.tmpdir(), ...)` and `envMixin['ZDOTDIR'] = zdotdir;`, the exact variant is found in AzureDataStudio, consistituting the Cross-Repo *Code Clone* variant.

New Vulnerabilities with Same Callee. The core *violation* of CVE-2025-20570 is that of sensitive data being stored inside temporary directory, which should be instantiated by `os.tmpdir()`. Grepping for `os.tmpdir()` yeild another discovery in VSCode, where the azure component tries to store the private key `cert.pfx` inside a temporary directory. Despite the effort of trying to delete the key after use, there exists a tiny time window where the attacker (another user in the system) can get the key value, thus the *abuse* still exists. In our testing, an adversary running a script in the background has an over 80% chance to conduct a successful attack.

8 Related Work

Graph-Based Static Analysis have been proposed for bug hunting in browsers [37,82,83]. These analyzers extract metadata during compilation, enabling targeted searches through manually written queries. Chucky [83] applied taint analysis to track user input and identify variables derived from the user input used without validation. Joern [82] introduced the code property graph, combining program properties, *e.g.,* ASTs and CFGs, for complex bug discovery. CodeQL [37] allows developers to embed expert knowledge within queries, facilitating the scanning of large codebases.

Fuzzing is an automated dynamic testing approach that explores programs by submitting thousands of malformed inputs per second [32, 51, 86]. It has proven effective in security-critical software systems [1,6,35,40,50,54,68,84,89,90,93]. As a security-critical application, browsers have also become a key target for fuzzing research, leading to the development of specialized fuzzers for different components, such as the DOM renderer [29,75,81,91,92], JS engines [42,77,78] and GPU libraries [3,63]. Despite their efforts, dynamic browser testing remain challenging because of the code complexity. Currently, only 29% of Chromium's codebase is covered by fuzzers and unit tests [25]. As a pattern-based static analyzer, GRAPE overapproximates to reduce FNs, and utilizing *Abuse* filtering to improve its precision.

Similarity-Based Bug Detection is a software engineering technique for identifying code-clone-related bugs [46, 47]. These approaches take a known vulnerability as input, generate a corresponding signature, and then match this signature against a program's signature library to identify potential buggy code segments. Although similarity-based detection and variant analysis share the same types of inputs and outputs, similarity-based techniques primarily focus on code-level similarity rather than the underlying root causes of vulnerabilities. As a result, these techniques can misclassify patched code as still vulnerable when the applied fix does not substantially alter the surrounding semantics. In contrast, GRAPE explicitly models the conditions that filters out superficially similar patterns that do not satisfy those conditions.

9 Conclusion

Web browsers expose a large attack surface to adversaries and therefore require protection. Complementary to formal verification, symbolic execution, and fuzzing, we propose GRAPE which systematizes and formalizes bug variant analysis, overcoming the challenges of code size, cross-context interactions, and cross-domain dependencies. GRAPE's *variant analysis workflow* narrows down code size, abstracts bug representation to resolve cross-context interactions, and finally simplifies cross-domain dependencies analysis using prior knowledges. We embed six patterns into our GRAPE prototype based on prior bug reports, and the prototype finishes the whole Chromium analysis within 12 minutes, discovering 24 new bugs, resulting in a total 17,500 USD bug bounty from Chrome VRP. Our discovered vulnerabilities impact web browsers like Chrome, and complex software system like OpenSSL, and VSCode.

Acknowledgments

We thank Qinying Wang, Daniele Cono D'Elia, the rest of the HexHive, and the anonymous reviewers for their detailed feedback. We also thank Google and Microsoft for acting quickly and professionally on our bug reports. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 18697, SNSF 10004669, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and an unrestricted gift from Google.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[3] Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. Darthshader: Fuzzing webgpu shader translators & compilers. *arXiv preprint arXiv:2409.01824*, 2024.

[4] Liu Bohan and Shi Haibin. Escape modern web-based app sandbox from site-isolation perspective. https://i.blackhat.com/Asia-24/Presentations/Asia-24-Liu-The-Hole-in-Sandbox.pdf, 2024.

[5] Liu Bohan and Wang Zheng. Reviving jit vulnerabilities: Unleashing the power of maglev compiler bugs on chrome browser. https://i.blackhat.com/EU-23/Presentations/EU-23-Liu-Reviving-JIT-Vulnerabilities.pdf, 2023.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[7] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578. IEEE, 2017.

[8] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A {Static/Symbolic} tool for finding good bugs in good (browser) code. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 199–216, 2020.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 18–35. IEEE, 2022.

[11] Chrome. Introduce of cpdfsdkformfillenvironment. https://pdfium.googlesource.com/pdfium/+/bf6f9b0692f3f09d768b047701733fa1b32c5918/, 2021.

[12] Chrome. Introduce of passwordaccepted. https://chromium.googlesource.com/chromium/src.git/+/23ba576ef5d8c0ac55e3df0243b12332152715e4, 2023.

[13] Chrome. Brp protected bugs no longer considered vulnerabilities. https://bughunters.google.com/about/rules/chrome-friends/5745167867576320/chrome-vulnerability-reward-program-rules, 2024.

[14] Chrome. Chromium bug tracker. https://issues.chromium.org, 2024.

[15] Chrome. Chromium code review policy. https://chromium.googlesource.com/chromium/src/+/lkgr/docs/code_reviews.md, 2024.

[16] Chrome. Chromium code unit tests. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/testing/testing_in_chromium.md, 2024.

[17] Chrome. Think about patterns. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/security-issue-guide-for-devs.md, 2025.

[18] chrome developer. Chrome memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/, 2024.

[19] chrome vulnerbility reporter. stack-use-after-return in gpu::gles2::programinfomanager::program::updatees2. https://crbug.com/40061249, 2022.

[20] chrome vulnerbility reporter. Security: Heap-use-after-free in ui::propertyhandler::getpropertyinternal. https://crbug.com/40065894, 2023.

[21] chrome vulnerbility reporter. Security: Stack-use-after-return in browserattestationservice::onchallengevalidated. https://crbug.com/40065577, 2024.

[22] chrome vulnerbility reporter. Uaf in lensoverlaycontroller:: lensoverlaycontroller. https://crbug.com/342419061, 2024.

[23] chrome vulnerbility reporter. Uaf in parsedarkcoloroverride. https://crbug.com/339788215, 2024.

[24] chrome vulnerbility reporter. use-after-free at browserusereducationservice.cc:120. https://crbug.com/340098902, 2024.

[25] Chromium. Chromium fuzzing code coverage. https://analysis.chromium.org/coverage/p/chromium, 2024.

[26] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[28] Chrome developer. Smart pointer guidelines. https://www.chromium.org/developers/smart-pointer-guidelines/, 2024.

[29] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.

[30] Alastair F Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. Industrial deployment of compiler fuzzing techniques for two gpu shading languages. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 374–385. IEEE, 2023.

[31] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, pages 10–10, 2020.

[33] Firefox. Moz2d. https://firefox-source-docs.mozilla.org/gfx/Moz2D.html, 2025.

[34] Firefox. Webrtc on firefox. https://blog.mozilla.org/webrtc/, 2025.

[35] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. {ACTOR}:{Action-Guided} kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5003–5020, 2023.

[36] Github. Codeql: About variant analysis. https://codeql.github.com/docs/codeql-overview/about-codeql/, 2024.

[37] GitHub. Codeql: the libraries and queries that power security researchers around the world. https://codeql.github.com/, 2024.

[38] Google. Chrome compilation time. https://groups.google.com/a/chromium.org/g/chromium-dev/c/kwfR0vwXmKg, 2023.

[39] google. Clusterfuzz. https://google.github.io/clusterfuzz/, 2023.

[40] Google. syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2025.

[41] Going Beyond Grep. Scaling variant analysis. https://goingbeyondgrep.com/posts/scaling-variant-analysis/, 2024.

[42] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

[43] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[44] Rong Jian and Guang Gong. Another way to talk with browser : Exploiting chrome at network layer. https://i.blackhat.com/USA-22/Thursday/US-22-Rong-Another_Way_to_Talk_with_Browser_Exploiting_Chrome_at_Network_Layer.pdf, 2022.

[45] Rong Jian, Leecraso, and Guang Gong. Put in one bug and pop out more: An effective way of bug hunting in chrome. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Leecraso_Put_In_One_Bug_And_Pop_Out_More_An_Effective_Way_Of_Bug_Hunting_In_Chrome.pdf, 2021.

[46] Wooseok Kang, Byoungho Son, and Kihong Heo. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1695–1708, 2022.

[47] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE symposium on security and privacy (SP)*, pages 595–614. IEEE, 2017.

[48] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[49] l4wio. Pdfium uaf series. https://docs.google.com/presentation/d/1TVohAIThGlwD8Mt3pp-8zayt7l5ArEDyWAsuQ4cRnJI, 2019.

[50] Gwangmu Lee, Duo Xu, Solmaz Salimi, Byoungyoung Lee, and Mathias Payer. Syzrisk: A change-pattern-based continuous kernel regression fuzzer. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1480–1494, 2024.

[51] libfuzzer. libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2023.

[52] Jungwon Lim, Yonghwi Jin, Mansour Alharthi, Xiaokuan Zhang, Jinho Jung, Rajat Gupta, Kuilin Li, Daehee Jang, and Taesoo Kim. Sok: On the analysis of web browser security. *arXiv preprint arXiv:2112.15561*, 2021.

[53] Linux. qspinlock: model checking benchmark. https://github.com/torvalds/linux/blob/master/kernel/locking/qspinlock.c, 2024.

[54] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX security symposium (USENIX security 19)*, pages 1949–1966, 2019.

[55] Lee Martin. Hello, webrtc on safari 11. https://leemartin.dev/hello-webrtc-on-safari-11-e8bcb5335295, 2017.

[56] Microsoft. Vscode. https://code.visualstudio.com/, 2024.

[57] Microsoft. Azuredatastudio. https://azure.microsoft.com/en-us/products/data-studio, 2025.

[58] Man Yue Mo. Variant analysis of web audio callback vulnerabilities in chrome. https://securitylab.github.com/resources/chrome_task_queue_uaf/, 2020.

[59] Tomáš Mráz. Avoid null dereference with pkcs7opsetdetachedsignature. https://github.com/openssl/openssl/pull/26078, 2024.

[60] Wang Nan and Xiao Zhenghang. Exploit chrome and firefox four times. https://i.blackhat.com/BH-US-24/Presentations/US24-Xiao-Super-Hat-Trick-Exploit-Chrome-and-Firefox.pdf, 2024.

[61] openhub. Chrome line of code. https://openhub.net/p/chrome/analyses/latest/languages_summary, 2024.

[62] OpenSSL. Openssl. https://openssl-library.org/, 2024.

[63] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Jing Tian, and Mathias Payer. {GLeeFuzz}: Fuzzing {WebGL} through error message guided mutation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1883–1899, 2023.

[64] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[65] Alexander Popov. Case study: Searching for a vulnerability pattern in the linux kernel. https://a13xp0p0v.github.io/2019/08/10/cfu.html, 2019.

[66] project zero. The more you know, the more you know you don't know. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html, 2022.

[67] project zero. The ups and downs of 0-days: A year in review of 0-days exploited in-the-wild in 2022. https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html, 2023.

[68] Zezhong Ren, Han Zheng, Zhiyao Feng, Qinying Wang, Marcel Busch, Yuqing Zhang, Chao Zhang, and Mathias Payer. Sysyphuzz: the pressure of more coverage. In *NDSS*, 2026.

[69] semgrep. Semgrep: Lightweight static analyzer for multiple languages. https://github.com/semgrep/semgrep, 2024.

[70] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[71] Skia. Welcome to skia: The 2d graphics library. https://skia.org/, 2025.

[72] Röttger Stephen. Breaking the chrome sandbox with mojo. https://i.blackhat.com/USA-22/Wednesday/US-22-Roettger_Breaking_the_Chrome_Sandbox_with_Mojo.pdf, 2022.

[73] Chromium Security Team. Chrome security checklist. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/checklist.md, 2024.

[74] Chromium Security Team. Chrome security faq. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/faq.md, 2024.

[75] the Project Zero team at Google. Domato: A dom fuzzer. https://github.com/googleprojectzero/domato, 2017.

[76] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391. IEEE, 2018.

[77] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. Dumpling: Fine-grained differential javascript engine fuzzing. In *NDSS*, 2025.

[78] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. {FuzzJIT}:{Oracle-Enhanced} fuzzing for {JavaScript} engine {JIT} compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1865–1882, 2023.

[79] Wireshark. Wireshark - coverage guided fuzzing benchmark. https://gitlab.com/wireshark/wireshark, 2024.

[80] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. Mitigating security risks in linux with {KLAUS}: A method for evaluating patch correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4247–4264, 2023.

[81] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[82] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[83] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013.

[84] Tingting Yin, Zicong Gao, Zhenghang Xiao, Zheyu Ma, Min Zheng, and Chao Zhang. {KextFuzz}: Fuzzing {macOS} kernel {EXTensions} on apple silicon via exploiting mitigations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5039–5054, 2023.

[85] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[86] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2013.

[87] Project Zero. Evaluating mitigations and vulnerabilities in chrome. https://security.googleblog.com/2024/10/evaluating_mitigations_vulnerabilities.html, 2024.

[88] Project Zero. Miracleptr: protecting users from use-after-free vulnerabilities on more platforms. https://security.googleblog.com/2024/01/miracleptr-protecting-users-from-use.html, 2024.

[89] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. Mendelfuzz: The return of the deterministic stage. *Proceedings of the ACM on Software Engineering*, 2(FSE):44–64, 2025.

[90] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1343–1360, 2023.

```
  // fix of CVE-2023-2932
  void CFFL_ListBox::SaveData() {
    m_pWidget->SetOptionSelection(i);
+    if (!observed_box) { return; } // fix
  }
  // new bug with similar caller
  void CFFL_ComboBox::SaveData() {
    ...
    else {
      m_pWidget->GetSelectedIndex(0);
      m_pWidget->SetOptionSelection(nCurSel);
    }
    // lacking validation
    m_pWidget->ResetFieldAppearance();
  }
```

Figure 16: Incomplete fix of CVE-2023-2932. GRAPE finds a Medium-Severity UAF and awarded 1K USD.

[91] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. Towards better semantics exploration for browser fuzzing. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):604–631, 2023.

[92] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: browser api fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1135–1147, 2022.

[93] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. Crossfire: Fuzzing macos cross-xpu memory on apple silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3749–3762, 2024.

[94] Jordy Zomer. Variant analysis of the 'sequoia' bug. https://pwning.systems/posts/sequoia-variant-analysis/, 2021.

A Appendix

A.1 Cross-Language Invalidation Variants

Following the pattern description in Section 6, we discover two new vulnerabilities in Chromium, involving the *Similar Caller* variant and the *Same Callee Chain* variant.

New Vulnerability with Similar Caller. CVE-2023-2932 is a cross-language invalidation found in class *CFFL_ListBox*. By leveraging the detection model, we notice that similar bug exists in its sibling class *CFFL_ComboBox*. Figure 16 illustrates our finding. The *SetOptionSelection* method can triggers an attacker controlled JS sequence, allowing the attacker to destroy

```
  // fix for crbug.com/40088733
   m_pWidget->ResetFieldAppearance(true);
+  if (observed_widget) { return; }
  // new bug with same callee chain
  m_pInteractiveForm->ResetFieldAppearance(...);
  m_pInteractiveForm->UpdateField();
```

Figure 17: Incomplete fix of *b/40088733*. GRAPE finds a Medium-Severity UAF and awarded 1K USD.

the *m_pWidget* object and subsequently exploit an UAF vulnerability. This discovery was initially introduced in March 2017, lasted for more than five years.

New Vulnerability with Same Callee Chain. Figure 17 shows a new vulnerability found by GRAPE. In the previous bug report, security researchers observed that *ResetFieldAppearance()* of *CPDFSDK_Widget* could potentially trigger a JS callback, allowing adversaries to free the widget. The developer therefore patched all code locations that called this function. However, manual analysis determines that class *CPDFSDK_Widget* ultimately calls *CPDFSDK_InteractiveForm*, implying the *ResetFieldAppearance()* method is also considered a violation function. GRAPE detects a violation where *CPDFSDK_InteractiveForm* is invoked and subsequently used without validation. This vulnerability was initially introduced in September 2021 [11] and remained hidden for the past three years.

A.2 Multiplication Overflow Variants

Using the pattern 4 in Section 6, GRAPE identifies ten new vulnerabilities derived across two reports. All reported issues have been confirmed as security vulnerabilities and subsequently patched, preventing the vulnerability being exploited.

```
// serve as an inspiration
I420Buffer::I420Buffer(width, height, stride_...)
: data_(AlignedMalloc(
    I420DataSize(height, stride_y, stride_u,
    stride_v)))
// new finding with similar caller
I210Buffer::I210Buffer(width, height, stride_...)
: data_(AlignedMalloc(
    I210DataSize(height, stride_y, stride_u,
    stride_v)))
```

Figure 18: High-Severity Integer Overflow in Chromium WebRTC. Detected by GRAPE.

New Vulnerability with Similar Caller. Figure 18 illustrates an integer overflow vulnerability that persists across similar caller implementations. In the *I420Buffer* class, the initialization process directly uses input arguments to compute *I420DataSize*, which can lead to an integer overflow when determining the allocation size. As a result, an insufficient

```
- pattern-either:
  - pattern: |
      $FREE_FUNC(...);
      if (!$WEAK_VAR)
        return ...;
  - pattern: |
      $FREE_FUNC(...);
      if ($WEAK_VAR) {
        $OTHER_FUNC(...);
        ...
      }
...
- metavariable-regex:
    metavariable: $WEAK_VAR
    regex: (^(weak_[a-z]+)$|...$)
```

Figure 19: Abbreviated violation query for bug variant 2, returning the list of all violation functions.

```
- pattern: $FREE_FUNC(...);
- metavariable-regex:
    metavariable: $FREE_FUNC
    # we put violation function name there
    regex: (^...$)
```

Figure 20: Abbreviated abuse query for bug variant 2, part of the abuse query to find all callsites of violation functions.

buffer is allocated, ultimately causing a heap-buffer overflow. GRAPE further observes that the sibling class *I210Buffer* inherits the same calculation method, directly operating input arguments without proper validation. Expanding this analysis, GRAPE identifies seven additional vulnerabilities exhibiting the same flaw. These findings were reported to Chromium and have been fixed. This decade-old code highlights GRAPE's ability to uncover long-overlooked vulnerabilities.

A.3 GRAPE Query Example

GRAPE requires *violation* and *abuse* queries that are leveraged together to search for bug candidates. While most variant patterns necessitates one-time manual efforts for *violation* identification, we observe that in some patterns the *violation* detection can be automated. The example here demonstrates how GRAPE integrates with SemGrep queries, focusing on bug variant 2 (UI destruction). The full list of queries and code is available in our prototype.

Figure 19 illustrates the process of extracting *violation* functions, where GRAPE identifies calls to *FREE_FUNC()* that are followed by a WeakPtr verification. All identified *violation* functions are exported and incorporated into the abuse pattern. Based on the generated list of *violation* functions, GRAPE locates all callsites of these functions as potential vulnerabilities (Figure 20). Subsequently, GRAPE identifies instances of safe *violation* handling, where calls are properly validated and do not result in abuse, as shown in Figure 21.

```
- pattern-either:
  - pattern: |
      $FREE_FUNC(...);
      if (!$WEAK_VAR)
        return ...;
  - pattern: |
      $FREE_FUNC(...);
      if ($WEAK_VAR) {
        ...
      }
# ...
- metavariable-regex:
    metavariable: $FREE_FUNC
    // we put violation function name there
    regex: (^...$)
```

Figure 21: Abbreviated abuse query for bug variant 2, part of the abuse query to filter safe uses.

```
// CVE-2024-0727
if (ctype_nid == NID_pkcs7_signed) {
+  if (p7->d.sign == NULL)  return 0;
  mdalgs = p7->d.sign->md_algs;
}

// unvalidated violation
case PKCS7_OP_SET_DETACHED_SIGNATURE:
  if (nid == NID_pkcs7_signed) {
    // p7->d.sign can be nullptr
    ret = p7->detached = (int)larg;
    PKCS7_type_is_data(p7->d.sign->contents);
  }
```

Figure 22: Incomplete fix of CVE-2024-0727. GRAPE finds an unvalidated violation, which was confirmed and fixed.

Finally, GRAPE removes the safe violation call sites identified in Figure 21 from the complete list of violation call sites (Figure 20), ultimately alerting the developer to only the remaining potential vulnerabilities.

A.4 Bug Discovery in OpenSSL

We evaluate the scalability of GRAPE on OpenSSL [62], one of the most security-critical cryptographic libraries [31]. Figure 22 highlights a bug discovered through this effort.

CVE-2024-0727 is a null pointer dereference vulnerability that allows a remote attacker to trigger a denial-of-service (DoS) condition on a server running an OpenSSL-based service. The vulnerability stems from the incorrect *assumption* that the structure `p7->d.sign` is always valid, *i.e.,* not a null pointer. When this *assumption* is violated, any access to members of `p7->d.sign`, such as `p7->d.sign->md_lags`, results in a null pointer dereference, leading to a potential crash and service disruption.

GRAPE identifies an unvalidated violation by analyzing variations along the same callee chain, treating `p7->d.sign`

```
// may destroy |this|
browser->Activate-ontents(...);
// access non-member function is safe
base::RecordAction(
  base::UserMetricsAction(...));
```

Figure 23: FP1: Object is not used after the UI destruction.

```
ReleaseProcess(); // free |this|
std::move(callback).Run(status); // on stack
if (weak_this) { ... } // protection
```

Figure 24: FP2: Object has been protected by WeakPtr.

as a key variable. It flags any usage of this structure that lacks proper null checks as potential *abuse* points. Following manual validation, we reported our findings to the OpenSSL maintainers, who acknowledged and patched the bug.

A.5 Full False Positives List

We conduct an in-depth analysis of FPs and categorize them into *four* primary types, as illustrated in Table 6. Specifically, among the 23 FPs, *ten* involve violation function calls that are already protected, *eight* have no abuse following the violation, *four* stem from incorrect identification of the violation function, and the remaining *one* is due to the object being protected by the C++ smart pointer. The comprehensive list of these FPs is available in Appendix Table 6.

No Abuse After Violation. For Unexpected UI Destruction, developers can safely access non-member functions even after `this` has been destroyed. As shown in Figure 23, while `ActivateContents()` may destroy the current class, the subsequent function call remains safe because `base::RecordAction()` does not interact with any destroyed class members or methods. Given Chromium's complexity, we opted not to craft rules that exclude such corner cases. However, this type of FP can be easily spotted by the user.

Protected Violation. Another frequent FP occurs when the code is already safeguarded by a weak pointer, yet GRAPE fails to detect this protection. In Figure 24, `ReleaseProcess()` may free `this`, but GRAPE incorrectly flags `std::move().Run()` as violating a safety assumption. However, since the callback function pointer is passed as a function argument (*i.e.,* a stack variable), the `weak_this` variable adequately ensures the object's temporal memory safety.

Not a Violation. The primary assumption for detecting Unexpected UI Destruction is that functions with the same name refer to the same function. While this assumption generally holds, cases arise where multiple functions from different classes share the same name, leading to FPs. As shown in Figure 25, GRAPE identifies a violation in `HandleKeyEvent()` at [1] and reports abuse at [2]. However, the identified violation

```
// [1] real violation function
delegate_->HandleKeyEvent(&key);
// calling HWNDMessageHandler::HandleKeyEvent
if (!ref) {return 0;}
// [2] reported: not a violation function
SuggestionStatus status = \
  current_suggester_->HandleKeyEvent(event);
// calling EmojiSuggester, not violation func
```

Figure 25: FP3: A function with the same name may not be a violation function.

```
// [1] unique ptr
auto owned_popup = base::WrapUnique(popup);
// [2] cannot destroy 'this'
owned_popup->DeleteDelegate();
```

Figure 26: FP4: Smart pointer prevents object destruction.

pertains to `HWNDMessageHandler::HandleKeyEvent()`, whereas the PF at [2] involves a call to `EmojiSuggester::HandleKeyEvent()`.

C++ Smart Pointer: C++ smart pointers are widely used in Chromium to manage object lifecycles [28]. Certain smart pointers, such as unique pointers, ensure that an object stays alive before it goes out of scope. As a result, the UI destruction model does not apply to these cases. For instance, in Figure 26, `owned_popup` is managed by a unique pointer, preventing its destruction until it goes out of scope; thus, [2] does not constitute a violation.

Overall, the observed FPs were easily identifiable with minimal manual effort. Therefore, we chose not to overfit our rules to improve the generality of GRAPE.

Table 6: List of all False Positives and their root causes. Locations are based on Chromium 130.0.6710.0.

ID	Violation Function	Code Location	FP Type
1	ActivateContents	chrome/browser/ui/tabs/saved_tab_groups/saved_tab_group_utils.cc:607	No Abuse After Violation
2	ActivateContents	chrome/browser/media/webrtc/media_stream_focus_delegate.cc:156	No Abuse After Violation
3	ActivateContents	chrome/browser/ui/views/extensions/extension_install_dialog_view.cc:215	No Abuse After Violation
4	ClientAreaSizeChanged	ui/views/win/hwnd_message_handler.cc:1746	No Abuse After Violation
5	ClientAreaSizeChanged	ui/views/win/hwnd_message_handler.cc:3040	Protected Violation
6	DeleteDelegate	ui/views/widget/widget.cc:2399	Protected Violation
7	DeleteDelegate	ui/message_center/views/message_popup_collection.cc:347	C++ Smart Pointer
8	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:256	No Abuse After Violation
9	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:416	No Abuse After Violation
10	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:1045	No Abuse After Violation
11	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:350	No Abuse After Violation
12	GeneratedPasswordAccepted	chrome/browser/ui/passwords/password_generation_popup_controller_impl.cc:271	Protected Violation
13	HandleKeyEvent	chrome/browser/ash/input_method/assistive_suggester.cc:404	Not a Violation
14	HandleMouseEvent	ui/views/win/hwnd_message_handler.cc:3276	Protected Violation
15	HandleMessage	chromecast/cast_core/runtime/browser/runtime_application_service_impl.cc:312	Not a Violation
16	OnDeviceScaleFactorChanged	ui/compositor/layer.cc:1480	Protected Violation
17	ReleaseProcess	content/browser/service_worker/embedded_worker_instance.cc:1046	Protected Violation
18	StopAnimating	ash/keyboard/ui/keyboard_ui_controller.cc:349	Not a Violation
19	StopAnimatingProperty	ui/compositor/layer.cc:431	Protected Violation
20	SyncWatchExclusive	mojo/public/cpp/bindings/lib/interface_endpoint_client.cc:696	Protected Violation
21	GetValue	third_party/pdfium/fpdfsdk/formfiller/cffl_textfield.cpp:90	Not a Violation
22	OnFormat	third_party/pdfium/fxjs/cjs_field.cpp:71	Protected Violation
23	quadCount	third_party/skia/src/gpu/ganesh/ops/FillRectOp.cpp:421	Protected Violation