USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Lancet: A Formalization Framework for Crash and Exploit Pathology

Qinrun Dai, Kirby Linvill, Yueqi Chen, and Gowtham Kaki,
University of Colorado Boulder

https://www.usenix.org/conference/usenixsecurity25/presentation/dai

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

Lancet: A Formalization Framework for Crash and Exploit Pathology

Qinrun Dai, Kirby Linvill, Yueqi Chen, Gowtham Kaki
University of Colorado Boulder
{qinrun.dai, kirby.linvill, yueqi.chen, gowtham.kaki}@colorado.edu

Abstract

Vulnerability and exploit analysis are at the heart of software security research and practice. However, a formalization framework for dissecting the cause, development, and impact of common software errors has been missing. To address this gap, we introduce `Lancet`, a formalization framework that reliably tracks three distinct types of ownership within its operational semantics that can be used to identify and differentiate between various types of vulnerabilities and exploit primitives *even in the presence of memory corruption*. Additionally, we developed two downstream tools, `FCS` and `EPF`, to demonstrate how security analysts can use `Lancet` for detailed crash and exploit analysis. `FCS` serves as a fast crash triaging tool, aiding patch synthesis in our system, which was selected as one of the winning teams in the DARPA AIxCC semi-final, while `EPF` fingerprints the transition of exploitation primitives to facilitate exploit analysis. Experiment results show that both tools are efficient and effective.

1 Introduction

Investigating and understanding vulnerabilities and exploits have long been fundamental to research and practice in software security, driving advances in root cause analysis, exploitability assessment, patch development, defense design and evaluation, and more. Yet, despite the many techniques developed and deployed, a formalization framework for systematically studying the causes, development, and impacts of vulnerabilities and exploits has been missing.

Current classification frameworks like the Common Weakness Enumeration (CWE) and ATT&CK catalog vulnerabilities in natural language, lacking the formality needed to provide analysis guarantees. Previous efforts have aimed to formally define vulnerabilities and reason about the presence or absence of bugs [3, 24–27, 44]. However, the program semantics proposed in these works are targeted at verifying program correctness or detecting vulnerabilities, where analyses stop once a bug is found, and cannot progress beyond a vulnerability. They are therefore not well-suited for the analysis of exploits where multiple types of security violations are often combined and chained.

To bridge this gap, we propose a formalization framework, `Lancet`, to formally define common memory corruption vulnerabilities, which account for over 70% of vulnerability fixes [38]. In addition to reasoning about concrete memory contents, `Lancet` reliably tracks three different types of ownership - cell owner, value owner, and pointee owner - that capture the mutual relationship between memory cells, in terms of operational semantics. These pieces of information are sufficient to capture both spatial and temporal memory safety violations, even in the presence of memory corruption. `Lancet` includes an extensible set of ownership [1] violation conditions to identify and distinguish between types of vulnerabilities.

To demonstrate how security analysts can leverage `Lancet` for dissecting crashes and exploits, we developed two downstream tools to showcase its utility. The first tool, `FCS`, is a fast crash triaging system designed to provide more reliable, accurate, and informative crash details than the widely used Address Sanitizer (ASan) [15, 33]. We deployed `FCS` in our winning Cyber Reasoning System (CRS) in the DARPA AIxCC semi-final [12] to prompt a Large Language Model to synthesize effective patches. The second tool, `EPF`, fingerprints the transitions of exploitation primitives to ease exploit analysis. `EPF` extends the vulnerability definitions in `Lancet` to formally define common exploitation primitives and techniques.

We evaluated `Lancet` and its two downstream tools `FCS` and `EPF`, using 116 test cases, including vulnerabilities from top-starred real-world programs and DARPA AIxCC semi-final challenge projects, and exploitation code from how2heap project, CTF challenge solutions, Juliet Test Suite [22], and public exploits. The evaluation results show that our tools can finish analysis for most cases in 20 seconds. In comparison with ASan, `FCS` avoids missing corruptions and reports correctly and informatively for almost all cases. `EPF` addition-

[1] Distinct from Rust ownership rules (See Section 3.1)

ally fingerprints primitive transition in exploits, a capability lacking in tools like ASan that are meant to analyze until a vulnerability is found. Further, FCS's more accurate and informative crash triaging effectively facilitates the synthesis of working patches.

In summary, this work makes three key contributions:

- Introducing a formalization framework, Lancet, for dissecting memory corruption vulnerabilities and exploits. See Section 3.
- Demonstrating the utility of Lancet through the development and application of two downstream tools for crash triaging and patch synthesis prompting, and exploitation primitive fingerprinting. See Section 4.
- Open-source implementation of Lancet and its downstream tools for x86, along with a comprehensive evaluation showing their efficiency and effectiveness. See Section 5 and 6.

2 Background and Related Work

In this section, we will discuss prior works on downstream applications that can be developed under our proposed Lancet and review related formalization efforts over the past decade.

2.1 Crash Triaging

The root cause of a program crash may be separated from the crash point by tens of thousands of instructions [40]. Thus, triaging tools generally fall into two categories: sanitizers, which report crash-related information in situ, and diagnostic tools, which are designed to locate and explain the root cause.

Sanitizers. Sanitizers are widely used by fuzzing tools due to their ability to detect the most common and exploitable types of corruption. Prominent sanitizers include Address Sanitizer (ASan) [15], Memory Sanitizer (MemSan) [16], and Undefined Behavior Sanitizer (UBSan) [8]. ASan and MemSan utilize poisoned redzones, instrumentation, and a runtime library, whereas UBSan detects undefined behaviors of C/C++ language by instrumenting heuristic checks.

While sanitizers are useful for detecting corruption, prior research [28, 35] has disclosed that their reports can sometimes be incorrect and misleading. Therefore, they cannot be reliably used for tasks that require more precision such as crash deduplication, reproduction, and manual debugging. Additionally, they offer limited insights into the root cause of crashes. We will further showcase this in our evaluation.

Root Cause Diagnosis. To reduce human efforts, tools have been developed to automate root cause analysis, such as POMP [40], RETracer [11], REPT [10], DEEPVSA [17]. These tools are based on reverse execution and backward taint analysis. For example, REPT and POMP analyze core dumps and Intel PT traces to highlight instructions that have data dependency with the crashing instruction. Therefore, naturally, they cannot diagnose situations where there is no direct data flow between the corruption site and the actual root cause, as discussed in Aurora [4]. Instead, Aurora first produces a diverse set of inputs and performs a statistical reasoning to synthesize predicates that describe the specifics under which a vulnerability is triggered, as explanation of root cause. Aurora's main drawbacks are the prolonged time needed to generate diverse inputs and the lack of guarantees about the accuracy of generated predicates. Igor [21] resembles Aurora by using a fuzzing procedure to minimize PoC's execution traces and prune test cases to extract the core behavior necessary to trigger the crash.

In contrast, our work Lancet lays out formal semantics and definition of vulnerabilities that can be used to provide guarantees for analyses. In the evaluation, we will showcase how its downstream tool FCS, a fast crash triaging system used during the DARPA AIxCC semi-final, provides reliable, accurate, and informative crash details.

2.2 Exploit Analysis

Vulnerability exploitation is generally a process of escalating primitives - a term to describe corruption capabilities achieved during an exploit. This process typically follows three steps. First, the exploit triggers a vulnerability, causing an initial corruption primitive. Then, this corruption is developed through manipulation techniques such as heap spraying [39], heap grooming [2], and stack pivoting [9] to tamper with sensitive data like function pointers. This step grants the exploit more advanced corruption primitives, such as control flow hijacking, invalid free, memory overlap, and arbitrary read and write. Finally, the exploit leverages these advanced primitives to bypass protections and persist attacks.

Despite this general workflow, each exploit's details are highly intricate due to the complexity and the diversity of different vulnerabilities. As a result, investigating exploit internals, particularly primitive transitions, is largely a manual process, as shown by Project Zero's blogs [42, 43] and white hat hackers' writeups [30].

Some automated exploit generation techniques incorporate limited fingerprinting elements for specific aspects of the workflow. For example, KOOBE [6] uses sanitizers' report to depict the corruption capability of a kernel vulnerability. SLAKE [7] applies SMT constraints to profile objects containing sensitive data. MAZE [37] models changes in heap layout using the Diophantine equation. HeapHopper [14] employs model checking to find weaknesses in heap implementations (*a.k.a.*, houses). ArcHeap [41] abstracts heap exploitation and uses heuristics to classify the impact of an exploit primitive.

So far, there is no formalization framework or downstream tool specially designed to identify exploitation primitives and fingerprint their transitions. Our work Lancet and its downstream tool EPF aim to address this gap.

2.3 Vulnerability and Exploit Formalization

The Common Weakness Enumeration (CWE) and ATT&CK frameworks provide a classification of vulnerabilities, but their descriptions are in natural language, lacking the formality needed to provide any analysis guarantees. Other work, such as the recent work into Incorrectness Separation Logic [24,31,32], provides formal descriptions of program incorrectness. This work can then be used to formally describe vulnerabilities, which are fundamentally incorrect program behaviors. These under-approximate formalizations can be used to prove no-false-positive theorems, ensuring that any detected vulnerabilities in a program are real vulnerabilities. However, they provide no guarantees that all vulnerabilities in a program are detected.

In contrast, other works, including Vellvm [44], SLAyer [3], and CompCert [26], focus on verifying or preserving program correctness, including the absence of vulnerabilities. For example, the semantics of Vellvm are used to verify the correctness of a compiler transformation that inserts bounds checks by verifying that no stuck state corresponding to a spatial memory safety violation can be reached [44]. Formalizations focused on program correctness support sound proofs of correctness, ensuring that any verified program does not contain (certain classes of) vulnerabilities. However, these over-approximate formalizations do not guarantee that all programs without vulnerabilities will be deemed safe, meaning they emit false positives.

The `LancetISA` semantics introduced in Section 3.3 are over-approximate with respect to ownership information, and therefore can similarly be used to formalize program correctness. However, the vulnerability rules introduced in Section 3.4 allow for detecting the presence of granular vulnerabilities including out-of-bound writes and use-after-frees, more akin to the fine-grained error definitions, such as null pointer exceptions, used in [24]. Regardless of whether prior works formally define vulnerabilities or program correctness, these works all rely on formal semantics that are only consistent until a corruption occurs. For example, Vellvm's semantics dictate that a double free causes the machine to enter a stuck state from which it cannot progress [44] while CompCert's semantics define free only in the case that the target memory has not yet been freed. Reasoning about exploits requires reasoning about program behavior after these kinds of corruptions occur. To the best of our knowledge, the `LancetISA` semantics are novel in their support for reasoning about program behavior even after corruptions occur.

3 `Lancet`- the Formalization Framework

In this section, we present `Lancet`, a formalization framework that reliably track three different types of ownerships of memory cells. We also define various types of vulnerabilities with CWE-IDs under this framework.

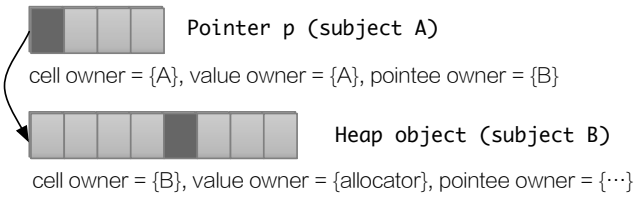


Figure 1: The cell owner, value owner, pointee owner of memory cells (in dark) hosting the pointer `p` (subject A) and the heap object (subject B) right after heap allocation.

3.1 Motivation

To start, we informally describe several common memory vulnerabilities and exploitation techniques, highlighting their shared characteristics and illustrating the core concept behind `Lancet`.

Out-of-bound access (CWE-125 and CWE-787) occurs when memory access exceeds the boundary of a memory object, a boundary that separates memory cells belonging to distinct subjects. A "subject" in this context is used to distinguish semantic chunks of memory which can be a stack variable, a global variable, a heap object, or a field in a structural variable. Uninitialized read (CWE-457) arises when a read memory cell contains a residual value from a different subject than the memory cell's owner subject. Use-after-free (CWE-416) happens when a dangling pointer is dereferenced. Initially, the pointer points to a memory cell owned by a specific subject, but before the pointer is dereferenced, the heap allocator reassigns this cell to another subject - either the allocator itself or another heap object if memory is already recycled. In this sense, there is no so-called "freed" memory; rather it is the subject's ownership of the memory that expires. The similar concept applies to stack-use-after-scope (CWE-562), where a stack variable is accessed after the function has returned. The commonality among these memory vulnerabilities lies in ownership violations.

From this perspective, the progression of an exploit is to amplify the initial violation introduced by a corruption vulnerability into increasingly powerful violations, by targeting memory cells with sensitive data. For example, in the *House of Einherjar*, an off-by-one overflow write first tampers with the metadata header of heap chunks with a NULL byte. This initial violation causes chunks to consolidate, creating a subsequent violation: the same memory cells within one chunk are simultaneously "owned" by an active subject and the heap allocator. This violation further escalates when the consolidated chunk is recycled to store heap objects containing function pointers, ultimately enabling an Address Space Layout Randomization (ASLR) bypass.

Our `Lancet` formalization framework is designed to identify these vulnerabilities and exploitation primitives by reasoning about ownership transfer and recognizing ownership violations. In `Lancet`, we differentiate among three types of

ownership. Figure 1 demonstrates this concept using heap allocation as an example. Informally, when a heap object is allocated, it becomes the **cell owner** of all memory cells that host the object - `cell owner = {B}`. However, the values stored in these cells still belong to the allocator or an expired subject - the **value owner**, as no new value has yet been written to the object. If the program attempts to read from these cells at this stage, it triggers an uninitialized read as the cell owner and the value owner are not consistent.

Further, the pointer to these memory cells, returned by `malloc`, can be stored in another set of cells. The **pointee owner** of the pointer's cells is the newly allocated heap object, indicating a reference relationship - `pointee owner = {B}`. Put another way, the pointee owner tracks the subject the pointer pointed to when it was created (or when the pointer is re-assigned for the sake of rigor). Intuitively, when a pointer accesses memory, the pointee owner of the pointer should be the same as the cell owner of the accessed cells. Otherwise, the pointer is referencing an incorrect, or stale, memory location.

Additionally, at any time, given a memory cell, it is supposed to have at most one cell owner, one value owner, and one pointee owner. Any deviation from this indicates that the cell is corrupted. In Section 3.4, we will detail the various causes of such corruption.

Note that, though `Lancet` and the Rust language both use the term "ownership", the concept differs between them. In Rust, the ownership is a language feature that programmers must adhere to for managing how values are accessed. Coupled with the borrow checker and the compiler enforced safety, Rust ensures that memory safety is not violated at the code level. The ownership in `Lancet`, however, is designed to define corruption situations, providing insights into the causes, development, progression, and impacts of vulnerabilities.

3.2 `LancetISA` Syntax

For the illustration purpose of this paper, we design a minimal register-register instruction set architecture, named `LancetISA`, to present how `Lancet` works. `LancetISA` is kept small to reduce the manual effort of formal modeling and analysis, yet more complex instructions can be emulated by composing multiple `LancetISA` instructions through transpilation, as we will show in our implementation (Section 5). The syntax of `LancetISA` is presented in Figure 2.

A program compiled to LancetISA consists of a sequence of instructions. The instructions operate using operands that can represent literal values (v), values stored in registers (r), or values stored in memory referenced by a pointer stored in a register ($[r]$). The `store` instruction stores a value from the source register (r) into the referenced destination memory ($[r']$). The `load` instruction loads a value from the source memory referenced by ($[r]$) to the destination register (r'). The `mov` instruction moves values into or between registers.

Common arithmetic and logical instructions such as `add` and `or` perform the given arithmetic or logical operation using the values from both register operands and store the result in the destination register operand (r'). The `cmp` instruction sets or clears the flag of a special register according to the result of the comparison operation. The `jeq` instruction jumps to the instruction at the (relative or absolute) address given by r if the flag in the special register is set. `push` and `pop` instructions add and remove values from the stack. The `LancetISA` supports arbitrary calls and includes specialized instructions to allocate (`malloc`) and deallocate (`free`) memory in the heap, assuming that the backend allocator correctly implements the desired functionality.

3.3 `LancetISA` Operational Semantics

Operational semantics of a language specify unambiguously how to execute a program written in that language. `Lancet` is equipped with a big-step operational semantics that specifies the effect each instruction has when evaluated. A big-step semantics, which naturally models each instruction as executing atomically, is given here rather than a small-step semantics, which allows for reasoning about concurrent and non-terminating executions, since it is a simpler form of semantics that is a sufficient foundation for the downstream tools introduced in this work which analyze concrete instruction traces after a sequential program has terminated. Reasoning about concurrent execution is discussed as future work in Section 7.

Most of the notation in the operational semantics is standard, but the notation $A[b//c]$ is used as shorthand for $A \cup \{b\} \setminus \{c\}$, *i.e.*, adding the element b to the set A while removing element c. The notation $a : a'$ is also used as shorthand for a range, so $B[a : a' \mapsto c : c']$ is shorthand for updating the map B so a maps to c, $a+1$ maps to $c+1, \ldots, a'-1$ maps to $c'-1$. The most interesting subset of the operational semantics rules is shown in Figure 3. The remaining rules are in Figure 14 for space considerations.

Each instruction produces a new state from a current state. The state is described by a concrete model of the register (R) and memory contents (M), a map tracking the size of allocated heap objects (H), a map tracking the subject a stack object was last allocated for (S), and ownership maps to track the cell owners (C), value owners (V), and pointee owners (P). The memory is flat and each cell has a unique address. There can be multiple owners (*i.e.*, $\mathcal{P}(\texttt{Subjects})$) for a memory cell to account for memory overlap during exploitation. Instructions are sequentially composed by executing against the state from the previous instruction, as described by rule SEQ.

Updates to register contents R and memory contents M are straightforward. Only `load` (resp. `store`) instructions can read from (resp. write to) arbitrary memory. Instructions follow the intel convention that instructions write to the first operand, e.g. `load` $r_l, [r_b]$ writes the contents of the address pointed to by r_b into the register r_l. The more interesting

		$r \in$ Registers $a \in$ Addresses $s \in$ Subjects $v \in$ Values		
x	$\in$	Operands	::=	v | r | $[r]$
bin	$\in$	Binary Instructions	::=	cmp | add | sub | imul | and | or | xor
i	$\in$	Instructions	::=	store $[r'], r$ | load $r,'[r]$ | mov r, v | mov r', r | $bin\ r', r$ | jeq r | push r | pop r | call r | malloc r | free r

Figure 2: Syntax of LancetISA - A register-register instruction set architecture.

parts of the semantics concern how Lancet soundly tracks ownership, even in the presence of corruptions.

Cell ownership is updated when a new atomic piece of data is created or destroyed. For the heap, creation is done through malloc, which allocates a new chunk of memory n bytes (cells) long as specified by the parameter r. The starting address of the user accessible part of the new chunk is stored in the special r_0 register while the heap size map H records the size of this allocated object. In the H map, only the starting address is mapped to the size of the object; all other cells default to $H = \perp$ since freeing memory is only valid when starting from the allocated address. The cell owner for the allocated memory range is updated to swap a fresh subject identifier created to represent this object in for allocator's special id allocator, thereby preserving other ownership information needed to track the downstream effects of exploits using overlapping objects. ArcHeap [41] shows that most modern allocators use an in-place metadata header for locality. In LancetISA, the cell owner of the header cells remains the special allocator subject in malloc. It allows to still capture memory safety violations, including writing outside the bounds of an allocated object, without needing to precisely model allocator-specific implementation details.

Dually free deallocates a memory chunk. The free object's size is read from the heap size map H. The cell owner for the freed memory range is swapped back to allocator to indicate that these cells are now re-managed by the allocator. Meanwhile, $H[a]$ is set to $\perp$ as a is no longer the starting address of an allocated object. It should be noted that, we only cancel the pointee owner of the pointer parameter in free in the cell owner set, ensuring that in continuous corruption where memory overlaps, like in the House of Einjerhar, we keep track of overlapping subjects.

Similar cell ownership semantics apply to stack data. A new stack variable is pushed onto the stack using the push instruction. The new variable is given a fresh subject identifier, which is swapped for the special stack id stack as the cell owner for the new stack data. The special sp register contains the address of the last object added to the stack, which grows from high address to low address. Since push both allocates and writes to memory, the value owner for the new stack data is set to the new subject. The pointee owner for the new stack data is likewise propagated from the value stored in the source register. Even if a pointer consists of multiple bytes, the pointee owner P only updates the mapping for the starting address of the new variable since the pointer is only valid to reference from that starting address. The fresh subject identifier is recorded in the map S to properly track which stack object is meant to be de-allocated when popped from the stack. The SUBSP rule in Figure 14 in the Appendix handles the creation of a stack frame through manipulating the sp register. It resembles the PUSH rule except that it does not update M, V, and P, as no value has been written to the stack yet. Stack data can be removed from the stack using the pop instruction. In this case the cell owner for the data replaces the last subject allocated on the stack ($S[a]$) with the special stack subject to indicate that the object has been de-allocated from the stack. Notably, since the data is not overwritten, the value and pointee owners remain unchanged, which is the same as the ADDSP rule in Figure 14.

In general, value ownership is updated whenever data is written to memory. In the STORE rule, the value owner of written memory cells is updated to the pointee owner of r_b - the register referring to the written memory. It reflects that the value flows to memory through the r_b pointer. If the pointer is not corrupted, the pointee owner of r_b should be the same as the cell owner of written memory. Otherwise, it indicates a corruption, which will be discussed in detail in Section 3.4.

For pointee ownership, since pointers may be copied between memory cells and registers several times before being used, the pointee ownership information needs to be propagated on any write to a register or a memory cell to properly track the subject the pointer originally referred to. For example, the STORE rule clears and updates the pointee owner of the written to memory cell with the propagated information from the source register. Likewise, the LOAD rule clears and updates the pointee owner of the destination register to ensure the pointee owner information is properly propagated through future use of the register. Additional related rules, MOVREGTOREG and MOVVALTOREG, are given in Figure 14. Collectively, these rules preserve ownership information even in the presence of aliasing.

Pointee ownership semantics is tricky in arithmetic instructions. For example, in p[10], the pointer p is used in an add instruction. If the index 10 is within the boundary, the pointee owner of the destination register can safely be updated as the same as the source register storing p. However, if the array has at most 10 elements, the resulting pointer actually crosses

$$\boxed{(\langle i\rangle, R, M, H, S, C, V, P) \Downarrow (R', M', H', S', C', V', P')}$$

$$\begin{array}{lll} R & \in & \texttt{Registers} \to \texttt{Values} \\ H & \in & \texttt{Addresses} \to \mathbb{N} \\ C & \in & \texttt{Addresses} \to \mathcal{P}(\texttt{Subjects}) \\ P & \in & \texttt{Registers} \cup \texttt{Addresses} \to \mathcal{P}(\texttt{Subjects}) \end{array} \qquad \begin{array}{lll} M & \in & \texttt{Addresses} \to \texttt{Values} \\ S & \in & \texttt{Addresses} \to \texttt{Subjects} \\ V & \in & \texttt{Addresses} \to \mathcal{P}(\texttt{Subjects}) \end{array}$$

$$\frac{(\langle i\rangle, R, M, H, S, C, V, P) \Downarrow (R', M', H', S', C', V', P') \qquad (\langle i'\rangle, R', M', H', S', C', V', P') \Downarrow (R'', M'', H'', S'', C'', V'', P'')}{(\langle i; i'\rangle, R, M, H, S, C, V, P) \Downarrow (R'', M'', H'', S'', C'', V'', P'')}\ \textsc{Seq}$$

$$\frac{n = R[r] \qquad a' = a + n \qquad fresh\ s}{(\langle \texttt{malloc r}\rangle, R, M, H, S, C, V, P) \Downarrow (R[r_0 \mapsto a], M, H[a \mapsto n], S, C[a : a' \mapsto C[a : a'][s//\texttt{allocator}]], V, P[r_0 \mapsto s])}\ \textsc{Malloc}$$

$$\frac{a = R[r] \qquad n = H[a] \qquad a' = a + n \qquad s = P[r]}{(\langle \texttt{free r}\rangle, R, M, H, S, C, V, P) \Downarrow (R, M, H[a \mapsto \bot], S, C[a : a' \mapsto C[a : a'][\texttt{allocator}//s]], V, P)}\ \textsc{Free}$$

$$\frac{w = sizeof(r) \qquad a = R[sp] \qquad a' = a - w \qquad fresh\ s \qquad C' = C[a' : a \mapsto C[a' : a][s//\texttt{stack}]]}{(\langle \texttt{push r}\rangle, R, M, H, S, C, V, P) \Downarrow (R[sp \mapsto a'], M[a' : a \mapsto R[r]], H, S[a' \mapsto s], C', V[a' : a \mapsto \{s\}], P[a' \mapsto P[r]][sp \mapsto P[a']])}\ \textsc{Push}$$

$$\frac{w = sizeof(r) \qquad a = R[sp] \qquad a' = a + w \qquad C' = C[a : a' \mapsto C[a : a'][\texttt{stack}//S[a]]]}{(\langle \texttt{pop r}\rangle, R, M, H, S, C, V, P) \Downarrow (R[sp \mapsto a'][r \mapsto M[a : a']], M, H, S[a \mapsto \bot], C', V, P[r \mapsto P[a]][sp \mapsto P[a']])}\ \textsc{Pop}$$

$$\frac{w = sizeof(r_r) \qquad a = R[r_b] \qquad a' = a + w \qquad v = R[r_r]}{(\langle \texttt{store}\ [r_b], r_r\rangle, R, M, H, S, C, V, P) \Downarrow (R, M[a : a' \mapsto v], H, S, C, V[a : a' \mapsto P[r_b]], P[a \mapsto P[r_r]])}\ \textsc{Store}$$

$$\frac{a = R[r_b] \qquad a' = a + sizeof(r_l) \qquad v = M[a : a']}{(\langle \texttt{load}\ r_l, [r_b]\rangle, R, M, H, S, C, V, P) \Downarrow (R[r_l \mapsto v], M, H, S, C, V, P[r_l \mapsto P[a]])}\ \textsc{Load}$$

$$\frac{r_l \neq sp \qquad v_l = R[r_l] \qquad v_r = R[r_r]}{(\langle bin\ r_l, r_r\rangle, R, M, H, S, C, V, P) \Downarrow (R[r_l \mapsto bin(v_l, v_r)], M, H, S, C, V, P[r_l \mapsto P[r_l] \cup C[bin(v_l, v_r)]])}\ \textsc{BinaryOperation}$$

Figure 3: The most interesting operational semantics of LancetISA. The remaining rules are in Figure 14 in Appendix.

a boundary. To not miss this error, the BINARYOPERATION rule doesn't clear the pointee owner of the destination register but conjuncts it with the cell owner of the arithmetic computation's results. As such, when a boundary is crossed, the destination register will have two pointee owners, indicating an ownership violation. Such a rule may inadvertently introduce over-approximation. For example, in `p=(p+q)/2`, where both `p` and `q` are pointers and `(p+q)` happens to be a valid memory address with a defined cell owner, the new `p` will thus have two pointee owners, even though the following memory access using the new `p` is within the boundary of `p` array. However, this over-approximation is actually a strength of Lancet: the memory safety of a program should not depend on the compiler's decision on memory layout - whether `p` array and `q` array are adjacent and if `p+q` is a valid address. Performing arithmetic on two pointers is a dangerous operation by any measure.

Though the BINARYOPERATIONS rule introduces over-approximation in the Pointee information, the concrete register and memory values remain precise. Specifically, the LancetISA semantics are precise with respect to register (R) and memory contents (M), heap objects (H), and stack objects (S). They are over-approximate with respect to ownership information including the cell owners (C), value owners (V), and pointee owners (P). In a bug, or vulnerability, finding setting, this over-approximation supports complete detection methods that ensure that any vulnerability that exists will be detected (no false-negatives). However, over-approximation cannot support sound detection methods where any reported vulnerability is guaranteed to be an actual vulnerability (no false-positives).

$$\boxed{\texttt{vuln} \in \mathit{Vulns}(\langle i \rangle, R, M, H, S, C, V, P)}$$

$$\frac{v_l = R[r_l] \qquad v_r = R[r_r] \qquad v_l' = \mathit{bin}(v_l, v_r) \qquad (P[r_l] = C[v_l] \ \wedge \ P[r_l] \neq C[v_l'])}{\texttt{OutofRangePointer(CWE-823)} \in \mathit{Vulns}(\langle \mathit{bin}\ r_l, r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{CrossBoundary}$$

$$\frac{\mathit{sizeof}(r_r) = w \qquad a = R[r_b] \qquad a' = a + w \qquad |P[r_b]| > 1 \ \vee \ (P[r_b] = C[a] \ \wedge \ P[r_b] \neq C[a' - 1])}{\texttt{OutOfBoundWrite(CWE-787)} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{OOBW}$$

$$\frac{a = R[r_b] \qquad P[r_b] \neq C[a]}{\texttt{ExpiredPointerDereference(CWE-825)} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{ExpiredWrite}$$

$$\frac{a = R[r_b] \qquad P[r_b] \neq C[a] \ \wedge \ \mathit{dom}(r_b) = \mathit{heap}}{\texttt{UseAfterFree(CWE-416)} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{UAFW}$$

$$\frac{a = R[r_b] \qquad P[r_b] \neq C[a] \ \wedge \ \mathit{dom}(r_b) = \mathit{stack}}{\texttt{StackUseAfterScope(CWE-562)} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{StackWriteAfterScope}$$

$$\frac{v_r = R[r_r] \qquad P[r_r] \neq C[v_r]}{\texttt{DanglingPtrOccur} \in \mathit{Vulns}(\langle \texttt{mov}\ r_l, r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{DanglingPtr1}$$

$$\frac{a = R[r] \qquad \texttt{allocator} \in C[a]}{\texttt{DoubleFree(CWE-415)} \in \mathit{Vulns}(\langle \texttt{free}\ \texttt{r} \rangle, R, M, H, S, C, V, P)}\ \textsc{DoubleFree}$$

$$\frac{\mathit{sizeof}(r_l) = w \qquad a = R[r_b] \qquad a' = a + w \qquad P[r_b] = C[a] \ \wedge \ P[r_b] = C[a' - 1] \ \wedge \ \exists\, i.\ a \leq i < a' \wedge C[i] \neq V[i]}{\texttt{UninitializedRead(CWE-457)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \textsc{UninitRead}$$

Figure 4: Example definition of vulnerabilities. The remaining is in Figure 15 in Appendix.

3.4 Definition of Vulnerabilities in `Lancet`

Generally, in `Lancet`, a memory vulnerability arises whenever a memory cell has more than one cell owner, or value owner, or pointee owner. For different vulnerability types, we further specify a function *Vulns* that uses the ownership information tracked by the semantics to identify vulnerabilities during a program's execution. Figure 4 presents the most notable definition rules. A more complete set is given in Figure 15 in the Appendix. Each conclusion in these rules follows the form $\texttt{vuln} \in \mathit{Vulns}(\langle i \rangle, R, M, H, S, C, V, P)$, which indicates that the vulnerability '`vuln`' occurs from executing the instruction i.

For example, the CrossBoundary rule formally defines one kind of spatial vulnerability that can arise from arithmetic operations on pointers: when the pointee owner of the destination register is the same as the cell owner of the pointed cell ($P[r_l] = C[v_l]$) but differs from the cell owner of the cell pointed to by the result of arithmetic operation ($P[r_l] \neq C[v_l']$). In other words, the pointer is manipulated to refer to another object not through assignment but via an arithmetic operation. Further, according to the BinaryOperation rule in the operational semantics, the destination register r_l will subsequently have more than one pointee owners. When this register is then used to store a value to memory, an out-of-bound access occurs, as specified by the OOBW rule.

Using these two rules, we find we can not only report an error earlier than ASan, but also illustrate how an initial cross boundary error develops into a out-of-bound access, identifying critical variables in this process. More importantly, unlike the definition of spatial safety in Vellvm [44] and HOTracer [20] discussed in the background (See Section 2.3), our definition doesn't assume that the pointer always points to the base address of an object because the pointee ownership accurately records the supposed target of the pointer. Therefore, downstream tools developed under `Lancet` are more precise and robust, even for fingerprinting exploits like the House of Einherjar, as we will show in the evaluation.

Another notable part of the premise in the OOBW rule is, besides checking the number of pointee owners, it also covers a corner case of out-of-bound access: a memory write spans multiple cells, extending from the end of one object till the start of another. It is achieved also by examining the consistence between pointee owner and cell owner.

For temporal safety, ExpiredWrite formally defines Expired Pointer Dereference (CWE-825) by checking if the pointee owner of the source register r_b is the same as the

cell owner of the referenced cell. This rule fundamentally differs from CROSSBOUNDARY and OOBW - their premises have no overlap. More specifically, instead of focusing on whether a pointer has been manipulated beyond a boundary or whether an access spans two objects, EXPIREDWRITE examines whether a previously valid pointer, without any pointer operation, is found to be pointing to an incorrect object. The only cause of this is that the cell owner of the pointed cell has been somehow altered, either by the allocator or stack prologue and epilogue.

Following EXPIREDWRITE, we can readily define a use-after-free write (CWE-416) by restricting the region domain of the address in $[r_b]$ to the heap (UAFW), and stack-write-after-scope (CWE-562) by restricting it to the stack (STACKWRITEAFTERSCOPE). We further generalize EXPIREDWRITE to DANGLINGPTR1 to identify the occurrence of dangling pointer before it is dereferenced.

Figure 4 further presents the formal definitions of two vulnerability types often overlooked in prior works. Double Free (CWE-415) is defined as an attempt to `free` a memory cell already owned by the special subject `allocator`. Uninitialized Read (CWE-457) is defined as a case where there is no out-of-bound read and no use-after free ($P[r_b] = C[a] \wedge P[r_b] = C[a']$), yet the cell owner and the value owner differ ($C[i] \neq V[i]$). In other words, the value stored in the cell is owned by a different subject, which commonly occurs immediately after allocation or function prologue.

The vulnerability detection rules rely on the over-approximate ownership information, meaning false-positives detections may occur. They are also non-exhaustive, meaning absence of detected vulnerabilities does not imply total memory safety. However, since the rules are over-approximate, if a vulnerability covered by the rules occurs during a sequential program execution, that vulnerability will be reported.

4 Downstream Tools

In this section, we demonstrate the utility of `Lancet` by illustrating how security analysts can leverage this framework to develop downstream tools.

4.1 Fast Crash Triaging

In the AIxCC semi-final, each team's Cyber Reasoning System (CRS) was required to automatically discover and patch vulnerabilities in the challenge projects (CP). To meet this objective, our CRS includes a fuzzing component to trigger crashes and a patching agent that synthesizes patches using a Large Language Model (LLM). To integrate these components, we developed a tool named `FCS`, which triages crashes to deduplicate reported errors and extract essential crashing context that can be embedded into LLM prompts.

Since the CRS was allocated for only four hours per CP, resource-intensive approaches like Aurora [4] and Igor [21] are impractical (Aurora can take up to 17 hours to run in certain cases). Other tools like POMP [40] and REPT [10] rely on Intel PT, which is not available in the dockerized environment of the competition. A straightforward approach would be to use sanitizer reports. However, prior research has shown that sanitizer reports can contain incorrect and misleading information [28, 35], making us hesitant to rely on them in a fully automated system. Furthermore, sanitizer reports are neither clear nor informative enough for effective patching, as we will demonstrate in the evaluation.

$$\frac{a = R[r] \qquad dom(a) = stack}{\texttt{HouseOfSpirit} \in Vulns(\langle \texttt{free r} \rangle, ...)} \text{ SPIRIT}$$

Figure 5: An example definition for the House of Spirit in `EPF`. Readers can refer to Figure 10 for more primitive definitions .

Given these considerations, we developed `FCS` under the `Lancet` framework. `FCS` first identifies the key variables that contribute to the satisfiability of a vulnerability definition rule in `Lancet`. It then retrieves the cell owners, value owners, and pointee owners of these key variables, annotating relevant instructions and mapping them to corresponding source code statements. `FCS` customizes its annotation focus based on the error type. For example, in cases of out-of-bound access, `FCS` concentrates on the code responsible for buffer allocation and index initialization. For use-after-free, `FCS` highlights where the dangling pointer is generated and whether it has aliases - pointers with the same pointee owner.

`FCS` is not designed as a sophisticated root cause diagnosis tool, but rather as an improvement over sanitizers. It aims to provide accurate and informative annotations of error causes and progression that are helpful for patch synthesis.

4.2 Exploit Primitive Fingerprinting

As discussed in Section 2.2, the essence of exploitation is to escalate primitives from the initial corruption to more powerful ones. One key task in investigating the internals of an exploit is to understand how primitives are obtained and transitioned. Currently, this fingerprinting process heavily relies on human efforts.

To this end, we developed a tool named `EPF` under the `Lancet` framework to facilitate the analysis of primitive transition. `EPF` extends `Lancet` to define common exploitation primitives and techniques. Figure 5 shows an example of the House of Spirit, a heap exploitation technique that remains effective in the newest glibc 2.40. It deliberately frees a stack memory chunk, resulting in a non-heap pointer being added to fastbin. Attackers can use the stack to manipulate heap content or vice verse. This technique can be identified by checking whether the domain of the `free` argument is stack or not. A more general form of the House of Spirit is invalid free, which is informally described as freeing any address that is

not a start of a heap object. Readers can refer to the Appendix for more definitions and explanation.

In Section 6.2, we will illustrate how EPF fingerprints the primitive transition from a limited off-by-one NULL overflow to use-after-free in the House of Einherjar. We will also show the results of applying EPF to analyze a real-world remote code execution in the Appendix.

EPF has certain limitations. For example, it cannot distinguish between restricted write and arbitrary write in a concrete execution context. This limitation inherits from the Lancet framework. In Section 7, we will discuss our future plans to symbolize Lancet to address these limitations.

5 Implementation

We have implemented Lancet for x86 and plan to extend it to ARM64 and RISC-V in the future.

Transpilation between LancetISA and x86. Instructions in x86 can be seamlessly transpiled to LancetISA. Data transfer instructions like mov, string manipulation instructions like movsb, and SIMD instructions like movaps are mapped to mov, store, and load instructions in LancetISA. Bitwise logic instructions like and and arithmetic instructions like add also have direct counterparts in LancetISA. The Load Effective Address (LEA) instruction, which is primarily used for pointer computations, array indexing, and accelerating integer calculations, is translated in LancetISA as a combination of bin and mov based on its scaled indexed addressing. Similarly, a comparison instruction in x86 is represented as a load followed by cmp and jeq. Beyond instruction transpilation, we developed specialized handlers for memory-related library functions such as memcpy and memmove, summarizing their effects and bypassing internal instructions to enhance performance.

To initialize ownership for global variables and other program assets like the Global Offset Table (GOT) and Procedure Linkage Table (PLT), we retrieve offsets and ranges using symbol tables, segment information, and the proc file. For stack frames, we trace each stack adjustment instruction, like sub rsp, 18h, to determine a function's usable stack range. Furthermore, for each local variable, we analyze source code to scale it within the stack frame according to compiler layout conventions. When source code is unavailable, we infer layout by examining stack addressing instructions to identify the start and end addresses of stack variables.

Deployment in AIxCC. FCS is developed as a downstream tool under Lancet framework for fast crash triaging, aiming to provide more reliable, accurate, and informative clues about the cause and progression of errors triggered by fuzzing. In our winning CRS for the AIxCC semi-final, we ran it as a micro-service before the patching agent in our pipeline.

The patching agent used GPT 4.0, the newest and strongest model from OpenAI before the semi-final. We adopted the recommendations from OpenAI user-guides [29] and designed the prompt engineering strategy in an interaction mode: GPT 4.0 takes the role of a "security expert", requesting guidance from a "user" role and views related code through a "tool" role. The FCS report is embedded into the "user" prompt.

Since Intel PT are not available in the AIxCC's dockerized environment, we leveraged Pin [19], a dynamic binary instrumentation tool, to collect instruction traces. Considering the overhead of Pin as a software-based approach, we selectively instrument up to 100,000 instructions before the corruption site. This threshold is safe based on our experiment results and prior research on root cause diagnosis [10, 40]. Besides, instructions in the fuzzing harness are skipped.

6 Evaluation

In this section, we evaluate Lancet by answering the following questions: (1) How efficiently can Lancet, and its two downstream tools FCS and EPF, dissect crashes and exploits? (2) Can FCS effectively annotate the cause and progression of errors, and can EPF illustrate primitive transitions in exploits? (3) Did FCS contribute to patch synthesis in the AIxCC semi-final? We will first present our experiment setup and then describe our experimental results.

Experiment Setup. We constructed a test case set from seven sources to cover a wide range of vulnerability types and exploitation techniques: (1) An internal dataset constructed during the preparation for the AIxCC semi-final. Cases in this dataset were pull issues of top-starred open source GitHub repositories. (2) Vulnerabilities in the Ngnix CP released by DARPA after the semi-final [13]. (3) Programs demonstrating heap exploitation techniques (*a.k.a.*, houses) from the how2heap project [34]. (4) CTF challenges with publicly available writeups and exploits. (5) Notorious public exploits in recent years, like Baron Samedit (*a.k.a.*, CVE-2021-3156 [23]). (6) Reproducible memory-corruption cases from REPT [10], used for efficiency comparison. (7) Representative test cases and variants drawn from the Juliet Test Suite [22], covering both memory-corruption and non memory-corruption CWE categories. For vulnerabilities from (1) and (2), we manually debugged them to obtain ground truth. These cases are listed in Tables 1, 2, and 5.

6.1 Efficiency

Tables 1 and 2 present the analysis times for FCS and EPF across crashes and exploits. They complete analysis within 25 seconds for 20 cases, with the longest duration extending to 201 seconds for 195 million instructions. We test them on Ubuntu 22.04, with 64GB RAM and 13th Gen Intel(R) Core(TM) i5-13600KF@3.50Ghz.

From the table, we observe that the time cost for PHP (CVE-2019-6977) and FFmpeg (10749) differs markedly, even though their trace lengths are similar. This is because of variations in the distribution of instruction types across

Test Case	ASan	FCS	Trace Length / LoC	Time (s)
Top-starred GitHub repos (crash)				
PHP, CVE-2007-1001 †	OOB ●	CROSSBOUNDARY	9M / 447K	18.17
PHP, CVE-2012-2386 †	OOB ●	CROSSBOUNDARY	10.7M / 676K	20.74
PHP, #76041 †	OOB ●	CROSSBOUNDARY	11.1M / 1M	22.57
PHP, CVE-2019-6977	missed ○	CROSSBOUNDARY	1695M / 2M	132.08
PHP, #16595	UAF ●	DANGLINGPTR1 Δ662	14M / 2M	63.33
GPAC, #2701	UAF ●	DANGLINGPTR4 Δ54	142M / 835K	7.29
GPAC, #2583	UAF ●	DANGLINGPTR4 Δ651	195M / 835K	201.44
Vim, CVE-2024-41965	DF ●	DANGLINGPTR1 Δ14	26M / 1M	31.08
Vim, CVE-2024-43374	UAF ●	DANGLINGPTR4 Δ441,687	26M / 1M	2.99
OpenSC, OSV-2023-1276	UAF ●	DANGLINGPTR4 Δ7	74M / 213K	13.87
Nasm, CVE-2004-1287 †	OOB ●	CROSSBOUNDARY	1.7M / 30K	2.91
mruby, OSV-2024-96	OOB ●	CROSSBOUNDARY Δ8	4M / 98K	9.03
FFmpeg, #10749	SEGV ○	CROSSBOUNDARY	1872M / 1.6M	17.34
FFmpeg, #11228	SEGV ○	NLLPTRDEREF1 UNTRUSTEDPTRDEREF1	31M / 1.6M	11.30
QuickJS, OSV-2024-204	UAF ●	DANGLINGPTR4 Δ112	1.9M / 86K	5.19
Nginx CP in the AIxCC semi-final (crash)				
CPV15	SEGV ○	UNTRUSTEDPTRDEREF1	40M / 177K	6.63
CPV10	UAF ●	DANGLINGPTR1 Δ10	40M / 177K	22.51
CPV5	SEGV ○	NLLPTRDEREF1 UNTRUSTEDPTRDEREF1	40M / 177K	21.96

Table 1: ASan and FCS on triaging crashes. ○ means ASan is either incorrect or misses the error. ◐ indicates ASan only reports the very initial corruption. Δ represents the number of instructions by which FCS reports the issue earlier than ASan. The length of a trace is measured in terms of the number of instructions. Cases with † symbol were also used in REPT [10].

cases. Especially, most state updates in Lancet are spent on MOV instructions which are memory involved, and the PHP trace contains roughly 737 million MOV instructions which is nearly five times the 150 million MOVs in the FFmpeg trace. A detailed distribution breakdown can be found in Appendix.

For comparison, we successfully reproduced four cases that were also evaluated in REPT [10]. The trace lengths we obtained for these cases are up to two orders of magnitude longer than those reported in REPT. The maximum analysis time we observed was 22.54 seconds, which remains within the same order of magnitude as REPT's reported per-instruction analysis time. This demonstrates that FCS achieves comparable efficiency to REPT. The time cost of other triage tools cannot be compared directly. For example, Igor [21], which employs a fuzzing procedure to extract core behaviors of a crash, requires at least 15 minutes, depending on the configured cut-off time. Similarly, Aurora [4] also incurs extra time on fuzzing.

Test Case	Tuned ASan	EPF	Trace Length / LoC	Time (s)
how2heap and CTF challenges (exploit)				
house_of_einherjar	OOB ◐	CROSSBOUNDARY OOBW INVALIDFREE OVERLAP UAFW	180K / 157	0.14
fastbin_reverse_into_tcache	UAF ◐	DANGLINGPTR4 SPIRIT	176K / 104	0.02
poison_null_byte	missed ○	EXPIREDWRITE EXPIREDWRITE OOBW OVERLAP	192K / 161	0.08
hacknote	UAF ◐	DANGLINGPTR4 UAF	178K / 160	0.12
Real-world exploits				
MiniDLNA, CVE-2023-33476	OOB ◐	CROSSBOUNDARY OOBW OVERLAP	1M / 25K	4.51
Sudo, Baron Samedit	OOB ◐	CROSSBOUNDARY OOBW OOBR	179K / 211K	0.07

Table 2: Tuned ASan's and EPF on fingerprinting exploits. The meaning of symbols is the same as Table 1.

6.2 Effectiveness

To evaluate the effectiveness of Lancet, we examine whether its downstream crash triaging tool, FCS, shows advantages in dissecting the cause and progression of crashes compared with its substitute target - ASan. Our evaluation criteria is the correctness and utility of the information in the reports. For all 14 crashes listed in Table 1, ASan missed one and provided incorrect vulnerability type information for four cases. In contrast, FCS not only successfully identifies all of them and provides correct crashing details, but also reports between 7 and 441,687 instructions ahead of ASan in 9 cases. Note that in FFmpeg #11228 and CPV5, a small offset from NULL address is dereferenced. While a manual scrutiny of the report content would reveal this, ASan reports them as simply SEGV without explicitly identifying them as NULL pointer dereference. FCS reports both NULL pointer dereference and untrusted pointer dereference (CWE-822) because the accessed memory cell is not defined with cell owner.

FCS demonstrates consistent effectiveness on the Juliet Test Suite. For 67 test cases in 18 memory-corruption CWEs, ASan produced 50 correct, 11 inaccurate, and 6 missed reports. In comparison, FCS generated 65 correct, 1 inaccurate, 1 missed reports. Taking CWE129_fgets_01 as an example, ASan fails to detect the error because the malicious input index falls outside of the redzone. FCS flags it by applying the CROSSBOUNDARY rule. We also evaluated 25 test cases from 8 non-memory-corruption CWEs. ASan produced 12 correct, 1 inaccurate, and 12 missed reports. In comparison, FCS generated 3 correct, 1 inaccurate, 21 missed reports. More details are in Table 5 moved to Appendix for space limit.

We also examine whether Lancet's other downstream tool, EPF, can fingerprint primitive transitions in exploits. Since no existing tool is specifically designed for this purpose, we use ASan as a makeshift solution, following the practice of KOOBE [6]. We tuned halt_on_error to 0 and set sanitizer-recover=address in ASan so that it won't immediately crash the program but activates a recovery mechanisms to continue reporting subsequent errors to the best of its ability after the initial corruption. EPF succeeds for all 6 exploits in Table 2, while ASan misses poison_null_byte and only reports the very initial corruption in the remaining 5 exploits.

In the following, we discuss two representative cases. A third case for how EPF fingerprints a remote code execution of CVE-2023-33476 is moved to Appendix due to space constraints.

CVE-2019-6977 in the PHP Interpreter. This case is a heap out-of-bound access error from Source (1). Figure 6 shows the vulnerable code snippet. In the function gdImageColorMatch of the file gd_color_match.c, a heap object is allocated on line 36 with an allocated size of 40 bytes. On line 41, the variable color is assigned a value that can be controlled by the input. If color is large enough, on line 43, the pointer bp will cross the boundary of buf, finally triggering an out-of-bound access on line 44.

In the reproduction of this vulnerability using the PoC from the PHP Bug Tracking System [36], ASan failed to detect it. Through extensive manual debugging, we discovered that the out-of-bound access was indeed triggered; however, the value 255 assigned to color in the PoC was excessively large, exceeding the red zone inserted by ASan.

To this end, we slightly decremented color to 248, at which point ASan detected an error but unexpectedly reported a "SEGV on unknown address". As shown in Figure 6, the report indicates that the crash occurs in zend_mm_del_segment which is totally irrelevant according to the ground-truth. After further manual investigation, we found that ASan actually failed to catch the initial out-of-bound access, thus allowing the error to propagate until it eventually caused an access to a high-value address.

We further adjusted color to 150, and finally, ASan detected the crash at the correct line in the correct function. However, the report still incorrectly identified the allocation site of the overflowed buffer. The buffer was actually allocated in gdImageColorMatch, but the report attributed it to gdImageCreateTrueColor. This mistake arises because the red zone accessed during the overflow belongs not to buf but to another object that happened to be allocated adjacent to buf.

Unlike ASan, FCS is more reliable, accurate, and informative. First, regardless of the value assigned to color, FCS always detects it, not at the overflow site on line 44, but one more statement earlier on line 43 where bp in rax just crosses the boundary. As in Figure 6, FCS applies the CROSSBOUNDARY rule, where P[rax] matches C[R[rax]] but differs from C[R[rax]+R[rcx]] (134 is the subject id of the object adjacent

Vulnerable Code of CVE-2019-6977

```
15 int gdImageColorMatch (…) {
17    unsigned long *buf, *bp;
19    int color;
36    buf = safe_emalloc(8, 5*im2->colorsTotal, 0);
41    color = im2->pixels[y][x];
43    bp = buf + (color * 5);
44    (*(bp++))++;
```

PoC

```
<?php $img1 = imagecreatetruecolor(0xfff, 0xfff);
$img2 = imagecreate(0xfff, 0xfff);
imagecolorallocate($img2, 0, 0, 0);
imagesetpixel($img2, 0, 0, 255); // "color" == 255
imagecolormatch($img1, $img2);          Evade ASan
```

ASan Report 1 ("color" == 248)

```
AddressSanitizer: SEGV on unknown address
  #0 0xc64a7a in zend_mm_del_segment /php-src/Zend/
zend_alloc.c:923:9                  Irrelevant Info
                . . . . . .
```

ASan Report 2 ("color" == 150)

```
AddressSanitizer: heap-buffer-overflow on address
0x7fffed61b7e0 …
READ of size 8 at 0x7fffed61b7e0 thread T0
  #0 0x896be5 in php_gd_gdImageColorMatch /php-src/
ext/gd/libgd/gd_color.c:44:13
                . . . . . .

0x7fffed61b7e0 is located 32 bytes to the left of
262144-byte region [0x7fffed61b800, 0x7fffed65b800)
allocated by thread T0 here:
  #0 0x4b243d in malloc (/php-src/sapi/cli/
php+0x4b243d)                       Incorrect Info
                . . . . . .
  #6 0x86828a in php_gd_gdImageCreateTrueColor /php-
src/ext/gd/libgd/gd.c:207:28
```

FCS Report

```
; 43: bp = buf + (color * 5)
0x668092: imul edx, [rbp+color's offset], 5
0x668096: movsxd rcx, edx
0x668099: shl rcx, 3
0x66809D: add rax, rcx
CrossBoundary: P[rax] = { "buf" }, C[R[rax]] = { "buf" } =>
C[R[rax]+R[rcx]] = {134}, C[rcx] = { "color" }
```

Figure 6: Tailored snippets of vulnerable code, PoC, ASan report, and FCS report for CVE-2019-6977 in the PHP Interpreter.

to buf at runtime), signaling that a pointer is manipulated to refer to a different subject. Second, FCS accurately identifies buf as the overflowed buffer by tracing where the subject in P[rax] was created. It also correctly reports color as the access index, according to the subject in C[rcx].

While FCS is not a sophisticated tool capable of explaining to human why color is too large or why buf is too small, it effectively annotates buf and color, the two most critical variables in an out-of-bound access. It meets our expectations for a fast crash triaging tool. As we will see in Section 6.3, these simple annotations can prompt LLM to synthesize root cause mitigation patches.

House of Einherjar. This heap exploitation technique, from Source (3), is widely-used and remains effective in the latest glibc 2.40. It is particularly useful because it can escalate the initial corruption, which is as limited as off-by-one NULL byte overflow, into a controllable use-after-free.

Figure 7 presents the change of cell owners of critical cells. For clarity, we retain only the core logic of the primitive transition, omitting non-essential statements. From the figure, we can observe that `EPF` successfully fingerprints each critical step in this primitive transition.

First, `EPF` locates the initial off-by-one and highlights that the cell owner of the overwritten memory cell is `allocator`, indicating a corruption of heap metadata. This corruption, along with the fake size prepared in `a[1]`, creates a fake chunk from the perspective of allocator. When `c` is freed on line 125, the allocator consolidates `a`, `b`, and `c` into a larger chunk, though `a` and `b` are still in use. Next, `EPF` accurately reports a memory overlap on line 130 when `d` is allocated. It further details that this overlap occurs between `a` and `d`, as well as `b` and `d`, demonstrating its robustness in analyzing the exploit. Further, `EPF` flags an invalid free on line 140, since the cell referred by the pointer `b` has more than one cell owners. By removing `b` from and adding `allocator` to the cell owner set, `EPF` ultimately reports a use-after-free on line 146. `EPF` shows that the corrupted cell is owned both by `allocator` and `d`, providing a qualitative description of controllability.

In contrast, ASan reports only a heap out-of-bound write on line 92, where the off-by-one is triggered, failing to detect all subsequent corruption behaviors. In fact, tools like ASan are fundamentally inadequate for exploit fingerprinting. This is partly because the runtime library introduced by ASan for detection purpose alters the glibc allocator's management logic. As a result, the chunk consolidation does not occur when ASan is enabled. Similar problem also exist in fingerprinting the exploitation of CVE-2023-33476 in Appendix.

Note that, the current `EPF` treats the allocator as a black box. Though it manifests changes in ownership, it lacks insight into how the allocator's internal design, behind `malloc` and `free`, drives these changes. In Section 7, we will discuss our future plan to open the black box to uncover finer details of primitive transition within the allocator.

6.3 Deployment in AIxCC

`FCS` was employed in our CRS for the AIxCC semi-final. It extracts essential crashing context that can be embedded into LLM's prompts for patch synthesis. We evaluate the usefulness of `FCS`, and thereby the `Lancet` framework, in this application scenario. Our evaluation criteria is the extent to which synthesized patches reassemble ground-truth patches written by human experts and address underlying root causes. We didn't validate patches based on whether PoCs could be reproduced after they were applied, as some synthesized patches only prevent the specific crash manifested by the PoC without rectifying the fundamental problem.

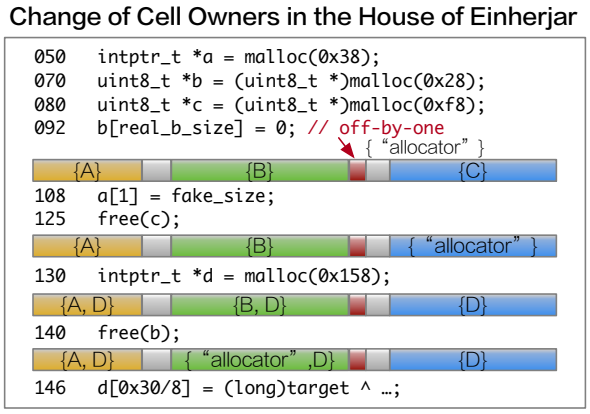


Figure 7: Change of cell owners in the House of Einherjar. We omit some noncritical details for simplicity.

Disclaimer. It should be emphasized that GPT 4.0 is not proven to possess true reasoning capability, and its working mechanism is not fully understood. In Section 6.2, we showcase that `FCS` is more reliable, accurate, and informative than ASan. However, we cannot conclude that this advantage will directly translate into `FCS` outperforming ASan in facilitating vulnerability patching, though our observations over a limited dataset suggest this. We look forward to future research works on interpretable LLM, which may yield further insights.

Patching CVE-2024-41965 in Vim. This is a double-free vulnerability in Vim collected in our internal dataset. Figure 8 shows the relevant code and the official patch [5]. The macro `VIM_CLEAR` frees the object referred to by its parameter pointer. When `buf->b_fname` and `buf->b_sfname` are aliases, subsequently executing `VIM_CLEAR` on them will trigger a double free. The patch rules out this alias situation.

ASan successfully detects this double-free (report details are in Figure 12 in Appendix due to space limit). The patch synthesized from the ASan report, as shown in the figure, adds a NULL pointer check which is actually redundant with the check inside `VIM_CLEAR`. GPT 4.0's "explanation" is "The key function in the stack trace is `dialog_changed` ... called multiple times." We infer that GPT 4.0 is misled by the repeated appearance of `dialog_changed` in the call stacks at both first and second free sites. ASan fails to indicate that `buf->b_fname` and `buf->b_sfname` are aliases.

In contrast, `FCS` detects this double-free vulnerability 14 instructions earlier, where the dangling pointer is used for arithmetic operation before being dereferenced. This is achieved by applying the ExpiredRead rule. Further, `FCS` extends `Lancet` to annotate double-free by searching for where the dangling pointer is created and whether there are other pointers sharing the same pointee owner. Therefore, `FCS` is able to report that `buf->b_fname` and `buf->b_sfname` are aliases. As

Official Patch of CVE-2024-41965

```
@@ -197,9 + 197,11 @@ dialog_changed () {
-   VIM_CLEAR(buf->b_fname);
+   // prevent double free
+   if (buf->b_sfname != buf->b_ffname)
+       VIM_CLEAR(buf->b_sfname);
+   buf->b_fname = NULL;
    VIM_CLEAR(buf->b_ffname);
-   VIM_CLEAR(buf->b_sfname);
```

Synthesized Patch using ASan Report

```
@@ -197,9 + 197,11 @@ dialog_changed () {
-   VIM_CLEAR(buf->b_fname);
-   VIM_CLEAR(buf->b_ffname);
-   VIM_CLEAR(buf->b_sfname);
+   if (buf->b_fname != NULL)
+       VIM_CLEAR(buf->b_fname);
+   if (buf->b_ffname != NULL)
+       VIM_CLEAR(buf->b_ffname);
+   if (buf->b_sfname != NULL)
+       VIM_CLEAR(buf->b_sfname);
```

Synthesized Patch using FCS Report

```
@@ -197,9 + 197,11 @@ dialog_changed () {
-   VIM_CLEAR(buf->b_fname);
-   VIM_CLEAR(buf->b_ffname);
-   VIM_CLEAR(buf->b_sfname);
+   if (buf->b_ffname != buf->b_fname)
+       VIM_CLEAR(buf->b_ffname);
+   else
+       buf->b_ffname = NULL;
+   if (buf->b_sfname != buf->b_fname &&
        buf->b_sfname != buf->b_ffname)
+       VIMC_CLEAR(buf->b_sfname);
+   else
+       buf->b_sfname = NULL;
```

Figure 8: The official patch, synthesized patch using ASan report and FCS report respectively, for CVE-2024-41965.

illustrated in Figure 8, the patch synthesized from the FCS report, though more verbose, is equivalent to the official patch. Readers can refer to Figure 12 in Appendix for report details.

Patching CPV15 in Nginx CP. This is a buffer overflow in Nginx CP, released by DARPA after the semi-final [13]. According to the competition log, our CRS was the only system to discover this vulnerability but failed to synthesize a working patch. This makes it an ideal case to understand the strengths and limitations of FCS.

This vulnerability is an internal overflow, from one subfield `uint32_t uid_got[4]` to another subfield `ngx_str_t cookie` in the `ngx_http_userid_ctx_t` structure, triggered in the function `ngx_decode_base64`. As ASan doesn't insert poisoned redzones within structure, it fails to detect the overflow initially. Instead, it reports a `SEGV on unknown address` later when the overwritten `u_char* data` in `ngx_str_t cookie` is dereferenced.

Figure 9 shows the official patch, which mitigates the overflow by limiting the length of the `ngx_str_t src` before calling `ngx_decode_base`. The patch synthesized using ASan report, however, mistakenly focuses on examining whether pointers, `conf->cookie` and `uid_data`, are NULL. GPT 4.0's "explanation" is "address the segmentation fault by ensuring pointers

Official Patch of CPV15 in Nginx CP

```
@@ -356,6 +356,18 @@ ngx_http_userid_get_uid (…) {
    ngx_str_t src;
    src = ctx->cookie;
+   src.len = 22;
   if(ngx_decode_base64(&dst, &src) == NGX_ERROR)
```

Snippet of Synthesized Patch using ASan Report

```
@@ -443,12 +443,18 @@ ngx_http_userid_set_uid (…) {
+   if(conf == NULL || conf->cookie == NULL) {
+       ngx_log_error(…);
+       return NGX_ERROR;
+   }
    uid.data = ngx_pnalloc(r->pool, uid.len);
    if(uid.data == NULL) {
+       return NGX_ERROR;
```

Snippet of Synthesized Patch using FCS Report

```
@@ -358,7 +358,12 @@ ngx_http_userid_get_uid (…) {
    src = ctx->cookie;
+   if (src.len > MAX_BASE64_INPUT)
+       src.len = MAX_BASE64_INPUT;
   if(ngx_decode_base64(&dst, &src) == NGX_ERROR)
```

Figure 9: The official patch, synthesized patches using ASan report and FCS report respectively, for CPV15 in Nginx CP.

are valid and memory allocations are checked". We infer that GPT 4.0 was misled by the SEGV message in ASan's report. Note that, sophisticated root cause diagnosis tools like POMP and REPT cannot handle this case well either, as they rely on reverse execution and backward taint analysis, making them unable to diagnose situation where there is no direct data flow between the corruption site and the actual root cause.

Unlike ASan, FCS reports that the segment fault occurs due to the untrusted dereferencing of `ctx->cookie.data` (CWE-822), with the last write in the function `ngx_decode_base64` (See Figure 13 in Appendix). GPT 4.0 interprets this as a "potential OOB write, leading to an invalid memory address". This patch synthesized using this information in the prompt attempts to ensure `src.len` is at most `MAX_BASE64_INPUT`. Unfortunately, `MAX_BASE64_INPUT` is not a valid macro in Nginx; it was instead fabricated by GPT 4.0, causing a compilation failure after applying the patch.

Though the patch synthesized using FCS report is closer to the ground-truth, the result reveals a limitation in the current FCS: it uses one single subject ID to represent the entire structure and fails to detect internal overflow. As a result, GPT 4.0 is left to infer potential OOB by using implicit information. In addition, the lack of information about the offset of `ctx->cookie.data` within `ngx_http_userid_ctx_t` forces GPT 4.0 to create a hypothetical `MAX_BASE64_INPUT` to cap the length. We plan to solve this problem in our future work.

Extended AI-Tool Comparison. To add a deeper AI-tool comparison, we extended our evaluation to Claude 3.7 and Grok3 for the two cases above—across three prompt configurations: (1) a baseline prompt enriched with UAF/SEGV knowledge, (2) UAF/SEGV knowledge augmented with ASan

reports and targeted hints, and (3) UAF/SEGV knowledge supplemented by FCS reports. We scored each model on reasoning accuracy and patch validation. Regardless of the LLM or prompt used, FCS consistently outperforms ASan in guiding the patch generation procedure. Detailed results are presented in Table 4 and discussed in the Appendix.

7 Limitations and Future Work

Support for Kernel and Other Architectures. The current implementation of `Lancet`, `FCS` and `EPF`, only supports userspace analysis. Our future work aims to extend support to kernel-specific artifacts. This will include transitioning privileged instructions to LancetISA, expanding the list of allocator interfaces by covering `kmalloc`, `kfree`, and other KPIs of SLAB/SLUB allocator, and defining kernel exploitation techniques like hijacking to `usermodehelper`. Besides, we plan to extent our architecture support to ARM64 and RISC-V.

Open the Blackbox of Allocator. Our current implementation treats the allocator as a black box, bypassing the analysis of instructions behind `malloc` and `free`. This simplification allows our design to be allocator-independent and enhances efficiency by avoiding the need to process a substantial number of instructions. However, this approach limits `FCS` in triaging allocator-specific errors and prevents `EPF` from disclosing finer details of primitive transition inside the allocator, for example, the consolidation of two chunks. To open the box, we will split a specific allocator into several subjects based on the metadata structure so that we can observe how they are corrupted internally.

Internal Overflow. Another problem in the current implementation of `FCS` is it considers a structural variable as a single subject without distinguishing between subfields. As a result, `FCS` fails to detect the internal overflow in CPV15 of Nginx CP. We will solve this problem by mapping the structure layouts to memory cells, provided that source code is available.

Concurrency and Symbolization. While the `Lancet` framework can analyze traces of concurrent software, it currently cannot triage concurrency bugs due to the absence of a concurrency model. We plan to develop such a model to extend `Lancet`. Besides, `Lancet` analyzes program in a concrete execution context. Therefore, we are not able to define and differentiate between primitives like restricted write and arbitrary write in `EPF`. Moving forward, we plan to symbolize `Lancet`, allowing the use of predicate logic to describe potential value stored in corrupted memory cells. This advancement will offer the opportunity to quantify exploitability. Specifically, we can define a partial order based on the number of corrupted memory cells as well as the entailment between potential value stored in them. It will allow us to formally determine whether exploitability is effectively escalated by an action, ultimately facilitating automate exploit generation.

More Applications. `FCS` and `EPF` are straight applications under the `Lancet` framework and they are limited. For example, `FCS` is designed for a rapid crash triaging, without aiming to diagnose the root cause. To enhance its capability, we plan to integrate algorithms from prior works, such as hypothesis testing in POMP, iterative analysis in REPT, and explanation synthesis in Aurora. In addition, we will make the output message more user-friendly, reducing the extra effort needed for interpretation. While `FCS` is efficient in triaging crashes, it is unsuitable for intensive fuzzing and thus was not used to discover new bugs in AIxCC. In the future, we plan to optimize it and explore its integration with AFL.

8 Conclusion

This work introduces `Lancet`, a formalization framework to dissect the cause, development, and impact of memory corruption vulnerabilities. The utility of `Lancet` is demonstrated through two downstream tools: `FCS`, a fast crash triaging tool and `EPF`, which fingerprints primitive transition. We implement them on x86 architecture and deployed `FCS` in the DARPA AIxCC semi-final to support patch synthesis. Experimental results validate the efficiency and effectiveness of these tools.

9 Open Science

To encourage further research in this area, we open source code of `Lancet` on GitHub [1] under the GPLv2 License. Additionally, we participated in the artifact evaluation process to validate the availability, reproducibility, and functionality of our work.

10 Ethics Considerations

This research involves a formalization framework to dissect the cause, development, and impact of memory corruption vulnerabilities. The downstream applications under this framework can better triage crashes for patch development and fingerprint primitive transitions for protection design. Our evaluation used reported and fixed vulnerabilities and well-known exploit artifacts. It was conducted in a controlled environment, ensuring that it does not impact external systems or environments.

11 Acknowledgment

We thank the anonymous reviewers for their valuable feedback, especially Reviewer A from our previous IEEE S&P submission for identifying critical typos in our semantic rules. We also acknowledge the DARPA AIxCC competition for inspiring the design of this work.

References

[1] Source Code. https://github.com/a85tract/Lancet.

[2] Azeria-lab. Grooming the IOS Kernel Heap . https://azeria-labs.com/grooming-the-ios-kernel-heap/, 2024.

[3] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 178–183, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium (USENIX Security)*, 2020.

[5] Christian Brabandt. patch 9.1.0648: [security] double-free in dialog_changed() . https://github.com/vim/vim/commit/b29f4abcd4b3382fa746edd1d0562b7b48c, 2024.

[6] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[7] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[8] Clang. UndefinedBehaviorSanitizer, 2024. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[9] CTFRecipes. Emerging 'Stack Pivoting' Exploits Bypass Common Security. https://www.ctfrecipes.com/pwn/stack-exploitation/arbitrary-code-execution/stack-pivoting, 2024.

[10] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[11] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[12] DARPA. AIxCC. https://aicyberchallenge.com/, 2024.

[13] DARPA. Nginx Challenge Project. https://github.com/aixcc-public/challenge-004-nginx-cp/tree/main/.internal_only, 2024.

[14] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[15] Google. AddressSanitizer, 2024. https://github.com/google/sanitizers/wiki/addresssanitizer.

[16] Google. MemorySanitizer, 2024. https://github.com/google/sanitizers/wiki/memorysanitizer.

[17] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. Deepvsa: facilitating value-set analysis with deep learning for postmortem program analysis. In *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security)*, 2019.

[18] hyprdude. chonked pt.1: minidlna 1.3.2 http chunk parsing heap overflow (cve-2023-33476) root cause analysis . https://blog.coffinsec.com/0day/2023/05/31/minidlna-heap-overflow-rca.html, 2023.

[19] Intel. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2024.

[20] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards efficient heap overflow discovery. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 989–1006, Vancouver, BC, August 2017. USENIX Association.

[21] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. Igor: Crash deduplication through root-cause clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[22] Juliet. *Juliet Test Suite v1.3 for C/C++ User Guide*. National Security Agency, 2017. Version 1.3.

[23] Himanshu Kathpal. CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit). https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit, 2021.

[24] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.

[25] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[26] Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363–446, December 2009.

[27] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1635–1648, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In *Proceedings of the 2022 Network and Distributed System Security Symposium (NDSS)*, 2022.

[29] OpenAI. Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering, 2024.

[30] Alexander Popov. Alexander Popov's Blog. https://a13xp0p0v.github.io/, 2023.

[31] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing.

[32] Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[34] Shellphish. how2heap. https://github.com/shellphish/how2heap, 2024.

[35] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[36] PHP Bug Tracking System. Sec Bug 77270: imagecolormatch Out Of Bounds Write on Heap. https://bugs.php.net/bug.php?id=77270, 2018.

[37] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security)*, 2021.

[38] White-House. Back to Building Blocks: A Path Toward Secure and Measurable Software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, 2024.

[39] Wikipedia. Heap spraying . https://en.wikipedia.org/wiki/Heap_spraying, 2024.

[40] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with Hardware-Enhanced Post-Crash artifacts. In *26th USENIX Security Symposium (USENIX Security)*, 2017.

[41] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *29th USENIX Security Symposium (USENIX Security)*, 2020.

[42] Google Project Zero. In-the-Wild Series: Android Post-Exploitation — googleprojectzero.blogspot.com. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-post-exploitation.html, 2021.

[43] Google Project Zero. 2022 0-day In-the-Wild Exploitation...so far — googleprojectzero.blogspot.com. https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html, 2022.

[44] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 427–440, New York, NY, USA, 2012. Association for Computing Machinery.

A Appendix

$$\frac{a = R[r] \qquad H[a] = \bot \vee |C[a]| > 1}{\texttt{InvalidFree} \in \mathit{Vulns}(\langle \texttt{free r} \rangle, \ldots)}\ \textsc{InvalidFree}$$

$$\frac{\begin{array}{c} a = R[r] \\ \exists\, i,\ a \le i < a+n,\ C[i] \neq \{\texttt{allocator}\} \end{array}}{\texttt{MemOverlap} \in \mathit{Vulns}(\langle \texttt{malloc r} \rangle, \ldots)}\ \textsc{Overlap}$$

Figure 10: More example definitions of common exploitation primitives and techniques in EPF.

More Example Definitions of Primitives and Explanation. Figure 10 presents two more example definitions of primitives. For invalid free, though the domain of memory cells to be freed could vary among stack (House of Spirit in this case), global, or heap (with Double Free being a subcase), their common characteristic is that H is marked as $\bot$. In the case of freeing the middle of a heap object, the cell owner of related cells won't be changed to the allocator, as the operational semantics of free in Lancet requires H to be defined. Therefore, when these cells are recycled in a sequential malloc, memory overlap is identified because a cell is co-owned by two subjects simultaneously.

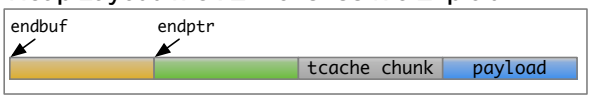
Remote Code Execution of CVE-2023-33476. This is a heap out-of-bound write in the HTTP chunk parsing code of the minidlna server, leading to remote code execution [18]. Figure 11 illustrates the heap layout after the heap grooming: a tcache chunk is positioned right after the endptr buffer, with the payload placed following this tcache chunk.

When memmove(endbuf, endptr, b->req_chunklen) is executed, the b->req_chunklen parameter is excessively large, triggering an overflow. As a result, the payload is moved to modify the tcache chunk, especially its fd pointer to point to the Global Offset Table (GOT). Then through spraying the tcache, the exploit tampers with the free entry in the GOT to ultimately hijack the control flow.

ASan reports the initial heap out-of-bonud read from endptr into its red zone but misses all subsequent errors, because its changes to the glibc allocator's management logic fails heap grooming. Even if heap grooming succeeded, the payload's overwrite of the tcache chunk would be noisily flagged as a use-after-free, as the corresponding shadow memory for the tcache chunk would record it as freed based on ASan's mechanism.

In contrast, EPF doesn't interfere with the exploit. Users can use cell owners of involved memory cells in its tracing

Heap Layout in CVE-2023-33476 Exploit



EPF Report

```
; 906: memmove(endbuf, endptr, h->req_chunklen);
CrossBoundary: P={ “endbuf” } => C={ “endptr” }
CrossBoundary: P={ “endbuf” } => C={ “allocator” }
CrossBoundary: P={ “endptr” } => C={ “allocator” }
CrossBoundary: P={ “endptr” } => C={ “payload” }
; 1129: h->req_buf = (char *)realloc(h->req_buf, …);
Overlap: C={ “GOT” , “h->req_buf” }
```

Figure 11: Heap layout after heap grooming in CVE-2023-33476 exploit and critical part of EPF report.

ASan Report for CVE-2024-41965

```
AddressSanitizer: attempting double-free on
0x0x502000019510 … in thread T0:
                . . . . . .
  #3 0x5555559625bf in dialog_changed /vim/src/
ex_cmds2.c:202:6
                . . . . . .
freed by thread T0 here:
                . . . . . .
  #2 0x5555559625bf in dialog_changed /vim/src/
ex_cmds2.c:202:6
```

FCS Report for CVE-2024-41965

```
0x5555557496fa: cmp word ptr [rax+98h], 0
ExpiredRead: P[buf->b_sfname] = {4576}, C[R[rax]+98h] =
{allocator}, P[buf->b_fname] = {4576}
```

Figure 12: Snippets of ASan report and FCS report for CVE-2024-41965. The 4576 is the ID of the subject referred by buf->b_sfname and buf->fname.

ASan Report for CPV15 in Nginx CP

```
AddressSanitizer: SEGV on unknown address
                . . . . . .
  #2 0x5555558c4f56 in ngx_htpp_userid_set_uid /src/
http/modules/ngx_http_userid_filter_module.c:446:13
                . . . . . .
```

FCS Report for CPV15 in Nginx CP

```
0x5555558c45e0: mov byte ptr [rax], cl ; in
ngx_decode_base64()
W to 0x555556264760, P = {ctx->cookie.data}
                . . . . . .
0x5555558c4f52: mov rcx, word ptr [rcx+0x28]
R from 0x555556264760, P = {ctx->cookie.data}
0x5555558c4f56: mov rdx, qword ptr [rcx]
R from 0xcccccccc
```

Figure 13: Snippets of ASan report and FCS report for CPV15 in Nginx CP during the AIxCC semi-final.

PHP, CVE-2019-6977			FFmpeg, 10749		
Opcode	**Count**	**Percent**	**Opcode**	**Count**	**Percent**
MOV	737,043,750	43.47%	MOVSS	188,480,226	10.34%
STOSB	235,221,642	13.87%	MOV	150,550,587	8.26%
ADD	202,827,926	11.96%	ADD	117,849,028	6.46%
MOVSXD	117,484,793	6.93%	STOSB	81,022,011	4.44%
LEA	101,910,312	6.01%	LEA	59,184,595	3.25%
MOVZX	84,862,336	5.01%	MOVSXD	43,031,408	2.36%
SHL	67,501,328	3.98%	ADDSS	39,807,324	2.18%
SAR	50,321,614	2.97%	SUB	35,177,523	1.93%
CMP	19,136,417	1.13%	SHL	30,885,691	1.69%
AND	17,255,163	1.02%	MOVAPS	28,819,934	1.58%

Table 3: Statistics of the most frequent selected instruction opcodes in the PHP and FFmpeg cases

record to understand how heap grooming is achieved. Further, as shown in Figure 11, EPF reports that the tcache chunk is overwritten on line 906, not because of a bogus use-after-free but because it is mistakenly treated as part of endbuf according to its pointee owner. In addition, when the GOT is overwritten on line 1129, EPF reports a memory overlap between it and the newly allocated h->req_buf.

Overhead and Instruction Distribution. We compared the four most frequent instructions (MOV, ADD, LEA, CMP) in PHP (1.059 B instructions) versus FFmpeg (326 M). With full analysis, PHP runs in 150 s (vs. 18 s baseline) and FFmpeg in 80 s (vs. 63 s). PHP's extra overhead is driven largely by MOVs (737 M vs. 150 M) and, to a lesser extent, LEAs (101 M vs. 59 M). Therefore, analysis time grows with instruction volume and handler complexity but isn't strictly linear.

Effectiveness of FCS and ASan on Juliet Test Suite. Table 5 in the Appendix summarizes our results on selected Juliet variants. For the heap-overflow case CWE129_fgets_01, ASan fails to detect the cross-boundary write: once the malicious input index falls outside ASan's red zone, no error is reported. In contrast, FCS always flags this CrossBoundary access by applying the ownership rule. For variants that perform arbitrary writes to legally addressable memory, neither tool reports an error, as no ownership invariant is violated. For non-memory-corruption variants, both ASan and FCS report as expected.

Discussion of AI-Tool Comparison. Table 4 presents the results of extending our evaluation to Claude 3.7 and Grok 3. While FCS 's report generally outperforms the ASan report, it fails to generate a valid patch for the CPV15 case due to its current single-ID struct limitation (Section 5.3). When provided only with UAF knowledge and ASan hints for the Vim case, Grok 3 succeeds in both reasoning and validation. We attribute this success to Grok 3's more recent training, which likely included the Vim case in its dataset.

$$\boxed{(\langle i\rangle, R, M, H, S, C, V, P) \Downarrow (R', M', H', S', C', V', P')}$$

$$\frac{}{(\langle \texttt{mov}\ r_l, r_r\rangle, R, M, H, S, C, V, P) \Downarrow (R[r_l \mapsto R[r_r]], M, H, S, C, V, P[r_l \mapsto P[r_r]])}\ \textsc{MovRegToReg}$$

$$\frac{}{(\langle \texttt{mov}\ r_l, v\rangle, R, M, H, S, C, V, P) \Downarrow (R[r_l \mapsto v], M, H, S, C, V, P[r_l \mapsto C[v]])}\ \textsc{MovValToReg}$$

$$\frac{\begin{array}{c} w = sizeof(addressbus) \qquad a = R[sp] \qquad a' = a - w \qquad ret = R[ip] \qquad fresh\ s \end{array}}{\begin{array}{l}(\langle \texttt{call r}\rangle, R, M, H, S, C, V, P) \Downarrow (R[sp \mapsto a'][ip \mapsto R[r]], M[a' : a \mapsto ret], H, S[a' \mapsto s], \\ C[a' : a \mapsto C[a' : a][s//\texttt{stack}]], V[a' : a \mapsto \{s\}], P[a' \mapsto P[r]][sp \mapsto P[a]][ip \mapsto P[r]])\end{array}}\ \textsc{Call}$$

$$\frac{a = R[sp] \qquad v_r = R[r_r] \qquad a' = a - v_r \qquad fresh\ s \qquad C' = C[a' : a \mapsto C[a' : a][s//\texttt{stack}]]}{(\langle \texttt{sub}\ sp, r_r\rangle, R, M, H, S, C, V, P) \Downarrow (R[sp \mapsto a'], M, H, S[a' \mapsto s], C', V, P[sp \mapsto P[a']])}\ \textsc{SubSP}$$

$$\frac{a = R[sp] \qquad v_r = R[r_r] \qquad a' = a + v_r \qquad C' = C[a : a' \mapsto C[a : a'][\texttt{stack}//S[a]]]}{(\langle \texttt{add}\ sp, r_r\rangle, R, M, H, S, C, V, P) \Downarrow (R[sp \mapsto a'], M, H, S[a \mapsto \bot], C', V, P[sp \mapsto P[a']])}\ \textsc{AddSP}$$

Figure 14: The remaining operational semantics rules of `LancetISA`.

LLM Patch Evaluation for CVE-2024-41965 with UAF knowledge and fix prompt																	
ASAN report						**ASAN report + Weak hint1**						`FCS` **report**					
GPT-4.0		**Claude3.7**		**Grok3.0**		**GPT-4.0**		**Claude3.7**		**Grok3.0**		**GPT-4.0**		**Claude3.7**		**Grok3.0**	
R	V	R	V	R	V	R	V	R	V	R	V	R	V	R	V	R	V
✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
LLM Patch Evaluation for CPV15 in Nginx CP with SEGV knowledge and fix prompt																	
ASAN report						**ASAN report + Weak hint2**						`FCS` **report**					
GPT-4.0		**Claude3.7**		**Grok3.0**		**GPT-4.0**		**Claude3.7**		**Grok3.0**		**GPT-4.0**		**Claude3.7**		**Grok3.0**	
R	V	R	V	R	V	R	V	R	V	R	V	R	V	R	V	R	V
✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗

Table 4: Comparative Evaluation of Foundation Models across Prompt Variations for Two Vulnerability Cases. R: Reasoning (analysis success / failure); V: Validation (functional correct / incorrect); *weak hint1*: rule out one mistaken fix—directly nullifying the pointer; *weak hint2*: a dereferenced pointer is not pointing to a valid address.

$$\boxed{\texttt{vuln} \in \mathit{Vulns}(\langle i \rangle, R, M, H, S, C, V, P)}$$

$$\frac{\mathit{sizeof}(r_l) = w \qquad a = R[r_b] \qquad a' = a + w \qquad |P[r_b]| > 1 \ \vee \ (P[r_b] = C[a] \ \wedge \ P[r_b] \neq C[a'-1])}{\texttt{OutOfBoundRead(CWE-125)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \text{OOBR}$$

$$\frac{a = R[r_b] \qquad P[r_b] \neq C[a]}{\texttt{ExpiredPointerDereference(CWE-825)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \textsc{ExpiredRead}$$

$$\frac{a = R[r] \qquad P[r] \neq C[a]}{\texttt{ExpiredPointerDereference(CWE-825)} \in \mathit{Vulns}(\langle \texttt{free r} \rangle, R, M, H, S, C, V, P)}\ \textsc{ExpiredFree}$$

$$\frac{a = R[r_b] \qquad P[r_b] \neq C[a] \qquad \mathit{dom}(r_b) = \mathit{heap}}{\texttt{UseAfterFree(CWE-416)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \text{UAFR}$$

$$\frac{a = R[r_b] \qquad P[r_b] \neq C[a] \qquad \mathit{dom}(r_b) = \mathit{stack}}{\texttt{StackUseAfterScope (CWE-562)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \textsc{StackReadUseAfterScope}$$

$$\frac{v_r = R[r_r] \qquad P[r_r] \neq C[v_r]}{\texttt{DanglingPtrOccur} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{DanglingPtr2}$$

$$\frac{v_l = R[r_l] \qquad v_r = R[r_r] \qquad (P[r_l] \neq C[v_l]) \ \vee \ (P[r_r] \neq C[v_r])}{\texttt{DanglingPtrOccur} \in \mathit{Vulns}(\langle \mathit{bin}\ r_l, r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{DanglingPtr3}$$

$$\frac{a = R[r_b] \qquad v = M[a] \qquad (P[a] \neq C[v])}{\texttt{DanglingPtrOccur} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \textsc{DanglingPtr4}$$

$$\frac{\mathit{NULL} < R[r_b] \qquad R[r_b] < \mathit{NULL} + 0\text{x}1000}{\texttt{NULLPtrDereference(CWE-476)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \textsc{NLLPtrDeref1}$$

$$\frac{\mathit{NULL} < R[r_b] \qquad R[r_b] < \mathit{NULL} + 0\text{x}1000}{\texttt{NULLPtrDereference(CWE-476)} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{NLLPtrDeref2}$$

$$\frac{a = R[r_b] \qquad C[a] = \{\}}{\texttt{UntrustedPointerDereference(CWE-822)} \in \mathit{Vulns}(\langle \texttt{load}\ r_l, [r_b] \rangle, R, M, H, S, C, V, P)}\ \textsc{UntrustedPtrDeref1}$$

$$\frac{a = R[r_b] \qquad C[a] = \{\}}{\texttt{UntrustedPointerDereference(CWE-822)} \in \mathit{Vulns}(\langle \texttt{store}\ [r_b], r_r \rangle, R, M, H, S, C, V, P)}\ \textsc{UntrustedPtrDeref2}$$

Figure 15: The remaining vulnerability definitions in `Lancet`.

Memory Corruption CWEs			
CWE	**ASan**	FCS	**Variant (Trace Length / LoC / Time(s))**
121	stack-buffer-overflow	OOB W CrossBoundary	CWE129_fgets_01◐(160.5K/168/0.05), CWE193_char_declare_loop_01(150.2K/111/0.02), CWE805_int_alloca_loop_13(147.7K/138/0.15), CWE805_int_alloca_loop_13_variant◐(147.7K/138/0.02) CWE805_struct_alloca_memmove_01(148.7K/107/0.17)
122	heap-buffer-overflow	OOB W CrossBoundary	CWE131_memcpy_66(152.8K/139/0.08), cpp_CWE129_fgets_09◐(1.8M/316/0.21), cpp_CWE193_char_cpy_01(1.8M/101/0.37), cpp_CWE805_char_loop_01(1.8M/112/0.26), cpp_dest_char_cat_01(1.8M/102/0.17)
123	SEGV	UNTRUSTED-PTRDEREF2	fgets_01◐(149.9K/137/0.05)
124	heap-buffer-overflow	OOB W CrossBoundary	malloc_char_loop_65(153.3K/167/0.03), malloc_char_loop_65_variant◐(153.3K/167/0.02), CWE839_fgets_01◐(152.5K/168/0.18), char_declare_cpy_01(148.4K/96/0.20), malloc_char_memcpy_01(152.2K/112/0.03)
126	heap-buffer-overflow	OOB R CrossBoundary	malloc_wchar_t_loop_66(154.5K/165/0.18), CWE129_fgets_01◐(152.9K/150/0.07), malloc_char_loop_01(154.5K/114/0.07), new_char_loop_01(1.8M/118/0.41), new_char_memmove_01(1.8M/108/0.28)
127	heap-buffer-overflow	OOB R CrossBoundary	malloc_char_loop_66(152.8K/171/0.12), malloc_char_loop_66_variant◐(152.8K/171/0.20), CWE839_fgets_01◐(152.5K/150/0.20), CWE839_negative_01(149.8K/126/0.10), char_alloca_memcpy_01(150.1K/100/0.10)
415	attempting double-free	DoubleFree	malloc_free_long_21 Δ106(149.5K/166/0.08), malloc_free_char_72 Δ6355(1.8M/189/0.28), new_delete_class_01(1.8M/105/0.28), no_assignment_op_01 Δ4(1.8M/187/0.10) new_delete_array_char_01 Δ2(1.8M/105/0.16)
416	heap-use-after-free	UAF R	return_freed_ptr_01 Δ8(151.9K/130/0.18), malloc_free_struct_01 Δ10(153.4K/128/0.18), new_delete_class_01 Δ1(1.8M/114/0.43), operator_equals_01 Δ36(1.8M/193/0.40) new_delete_array_struct_01 Δ10(1.8M/132/0.48)
457	Missed	UninitRead	double_array_alloca_partial_init_01(175.4K/133/0.55), double_array_malloc_no_init_01(167.0K/124/0.35), twointsclass_array_malloc_partial_init_01(167.2K/129/0.27), empty_constructor_01(1.8M/119/0.30) char_pointer_01(149.3K/90/0.15)
476	SEGV	UNTRUSTED-PTRDEREF1 NllPtrDeref1	long_66(147.1K/162/0.20), binary_if_01(147.1K/81/0.27), class_01(1.8M/111/0.25), deref_after_check_01(147.1K/79/0.28), struct_01(147.1K/101/0.28)
587	SEGV	UNTRUSTED-PTRDEREF1	basic_06(147.1K/106/0.10)
590	attempting free on address which was not malloc()-ed, stack-use-after-scope	INVALIDFREE	delete_array_char_static_01(1.8M/98/0.40), delete_array_class_declare_01(1.8M/110/0.14), delete_array_class_static_01(1.8M/110/0.20), delete_char_placement_new_01(1.8M/97/0.30), int_static_01(1.8M/96/0.28)
665	Missed	Missed	char_cat_01(151.1K/94/0.22)
672	heap-use-after-free	UAF R	list_int_54a Δ304(1.8M/379/0.50)
680	heap-buffer-overflow	OOB W CrossBoundary	malloc_fgets_01◐(154.6K/117/0.15), malloc_fixed_01(10737.6M/106/0.01), new_fixed_01(10739.2M/110/0.01), new_fgets_01◐(1.8M/121/0.25)
758	SEGV	UNINITREAD	int_malloc_use_06(151.6K/120/0.27), char_alloca_use_01(149.8K/78/0.40), char_malloc_use_01(151.7K/82/0.25), char_new_use_01(1.8M/87/0.38), char_pointer_new_use_01(1.8M/87/0.30)
761	attempting free on address which was not malloc()-ed	INVALIDFREE	wchar_t_console_66(152.0K/204/0.07), char_fixed_string_01(149.8K/107/0.20)
789	heap-buffer-overflow	OOB W CrossBoundary	malloc_wchar_t_fgets_01(154.1K/174/0.20), new_wchar_t_fgets_01(1.8M/178/0.36)
Non Memory Corruption CWEs			
134	SEGV	UNTRUSTED-PTRDEREF1/2	char_console_fprintf_01◐(153.6K/143/0.21)
190	Missed	Missed	char_max_multiply_01(149.8K/109/0.21), char_max_add_01(149.8K/104/0.33), char_max_square_01(149.8K/106/0.08)
252	Missed	Missed	char_fgets_01(151.6K/91/0.04), char_fread_01(150.7K/86/0.10), char_putchar_01(149.3K/74/0.05), char_rename_01(149.6K/83/0.10), char_putc_01(149.9K/74/0.12)
401	detected memory leaks	Missed	char_calloc_01(150.5K/103/0.22), char_malloc_01(150.5K/103/0.13), char_realloc_01(150.6K/103/0.01), new_TwoIntsClass_01(1.8M/117/0.23)
562	stack-use-after-return	STACKREADUSE-AFTERSCOPE	return_buf_01(148.9K/82/0.08), return_local_class_member_01(149.8K/107/0.15), return_pointer_buf_01(149.3K/88/0.15)
667	Missed	Missed	basic_01(156.5K/93/0.08)
690	Missed	Missed	char_calloc_01(150.6K/86/0.23), char_malloc_01(150.6K/86/0.17), int64_t_realloc_01(151.2K/86/0.11)
762	alloc-dealloc-mismatch	Missed	calloc_delete_01(1.8M/185/0.33), delete_array_char_malloc_01(1.8M/104/-0.03), new_array_delete_int64_t_01(1.8M/102/0.27), new_delete_array_class_01(1.8M/102/0.20), new_free_class_01(1.8M/103/0.28)

Table 5: ASan and FCS on Juliet Test Suite [22]. The "Variant" names drop the CWE prefix—for example, CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_01 → CWE129_fgets_01. The meaning of symbols is the same as Table 1.