Changes in the Guard: Detecting Authority-Exposing Patches with Differential Guarded Flow Paths

Nathan Dautenhahn
Dartmouth College
Hanover, New Hampshire, USA
Serenitix
Houston, Texas, USA
Nathan.D.Dautenhahn@dartmouth.edu

Sergey Bratus
Dartmouth College
Hanover, New Hampshire, USA
Sergey.L.Bratus@dartmouth.edu

Abstract

Software changes constantly, and patches keep introducing vulnerabilities. Many of them have one shape: a patch removes or weakens a check, and an attacker gains authority that was previously out of reach. The question an analyst asks of such a diff is not "is there a bug?" but "what can I *do* now that I could not before, and along which path?" Bug detectors match shapes in one version; taint trackers report that data reaches a sensitive operation but keep the check beside the query, not in the program. Neither can name the check a patch removed.

We introduce **Guarded Flow Paths (GFP)**, in which every value flow carries the branch conditions under which it happens, so a check is a label on the path from an input to the operation that uses it. A checker names a source, a sensitive operation, and the condition under which that operation becomes a violation: the same integer may select an address, a write length, or a record, and the operand it reaches says which. One reachability query answers every checker. *Differential authority analysis* runs that query on both sides of a patch and reports the authority gained or lost: which sensitive operations an input can now reach, and which guard protected each path before. Our intra-procedural prototype, AuthDiff, lifts LLVM IR into GFP; on a worked example it reports nine newly unconditioned source/endpoint pairs with thirteen witness paths, and seventeen written checkers span CWE and OWASP classes on synthetic examples and one OpenSSL excerpt.

CCS Concepts

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Automated static analysis*; *Software reverse engineering*.

Keywords

patch analysis, authority analysis, guard removal, authority, value-flow, program dependence, reverse engineering, vulnerability research, software understanding

ACM Reference Format:
Nathan Dautenhahn and Sergey Bratus. 2026. Changes in the Guard: Detecting Authority-Exposing Patches with Differential Guarded Flow Paths. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3847158.3847387



This work is licensed under a Creative Commons Attribution 4.0 International License.
SURE '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3019-1/2026/11
https://doi.org/10.1145/3847158.3847387

1 Introduction

Software changes constantly, and a small patch can change what an attacker can do. Removing a bounds check, relaxing a validation, or dropping an authorization step can expose a sensitive operation to inputs that previously could not reach it, whether by accident or by design [2, 3]. For an exploitation researcher triaging a patch—or an *n*-day analyst reversing a security fix—the central question is:

> *What can an attacker do after this change that they could not before, and along which path?*

Today's tools find *that* data reaches an operation; none records *under what condition* it got there. Bug detectors such as Infer Pulse [4], the Clang Static Analyzer [7], and Coverity [6] match a bug shape, null dereference or use-after-free, at the operation itself. Taint analyses such as Semgrep [8], CodeQL [5], and Infer's taint mode report that data flows from a source to the operation, but the check that made that flow safe is not part of what they track; it sits beside the query. So neither can say which check a patch removed.

Which values are legitimate depends on what the developer intended, and an analyst rarely knows that. What an analyst can know is what an operation requires of its operands: a bound on a `memcpy` size, an ownership test on a document identifier. Put the requirement on the operand rather than the variable, and many vulnerability classes take one shape: an input reaches a sensitive operation on a path where the check it requires never occurs.

We argue the gap is in what the analysis records. To see that a check disappeared, every value flow must carry the condition that guarded it. **Guarded Flow Paths (GFP)** do that: each flow edge is labelled with the branch condition that governs it, so a path that became unconditional across a patch is a difference between two graphs. On top of GFP, a *checker specification* names the sources, the sensitive operation, and the condition under which that operation becomes a violation, and the analysis is one query: is there a path from a source to the operation on which that condition holds? What an input reaches through those operations is its *authority* [14]. *Differential authority analysis* runs the query on both sides of a patch and subtracts; what remains is the operations that lost their check, each with the path that reaches it and the guard that used to stand on that path.

Our intra-procedural prototype, AuthDiff, lifts LLVM IR into GFP and implements the queries, the comparison, and witness reporting. We study synthetic examples and one OpenSSL excerpt

with written checkers for seventeen classes. On the worked integer example (§6.1), AuthDiff reports nine newly unconditioned source/endpoint pairs and thirteen witness paths. Infer, the Clang Static Analyzer, and Semgrep were run directly; CodeQL and Coverity are assessed from documentation (§8).

Contributions.

- **Guarded Flow Paths** (§4): a value-flow graph whose edges carry their guards, so a removed check is a structural fact.
- **Checker specifications** (§6): sources, a sensitive operation, and its violation condition; one query covers seventeen classes.
- **Differential authority analysis** (§5): the query on both sides of a patch, reporting the operations that lost a guard, with path and guard as evidence.
- **AuthDiff and its evaluation** (§7, §8): an intra-procedural prototype, localized examples, and a five-tool comparison.

2 Motivating Example

Programs routinely receive values that are unsafe for some uses. A pointer may be null, a length may exceed the available buffer, or a document identifier may refer to an object the caller is not permitted to access. Safety therefore depends on the conditions under which the program uses a value. A non-null check can protect a dereference; a bounds check can protect a copy; an ownership check can protect document retrieval.

These protections often take the form of guards: the sensitive operation remains in the program, but the guard restricts when execution can reach it. Removing the guard can expose the operation even when the operation and its inputs remain unchanged; for a patch, the question is whether the change removed the check that operation requires.

Consider a synthetic document handler whose policy permits a caller to retrieve only documents they own. The handler accepts a document identifier, checks ownership, and then fetches the document. A patch removes the check:

```
// Before
if (!acl_check(user_id, doc_id)) {
    return NULL;
}
return db_fetch_doc(doc_id);

// After
return db_fetch_doc(doc_id);
```

The prototype lifts each version into the guarded form below (abbreviated), in which every definition carries the condition under which it happens; `@ true` means unconditional, and any other label names the value that must hold:

```
// Before
params: %user_id, %doc_id
def %allowed <- call(@acl_check, %doc_id, %user_id)  @ true
def %is_ok   <- icmp(const:0, %allowed)              @ true
def %doc     <- call(@db_fetch_doc, %doc_id)         @ %is_ok
def ret      <- ret(const:null, %doc)                @ %is_ok

// After
params: %user_id, %doc_id
def %doc     <- call(@db_fetch_doc, %doc_id)         @ true
def ret      <- ret(%doc)                            @ true
```

The document identifier reaches the same fetch in both versions. Before the patch, the caller must pass the ownership check for that document: the fetch is defined only `@ %is_ok`. Afterward, the caller can select a document without passing that check. The authority the patch exposes is this newly unrestricted fetch, not a new operation or a changed type.

A checker for this case names the caller-supplied document identifier as the source, the fetch as the sensitive operation, and the ownership condition required for that user and document. A violation occurs when the identifier reaches the fetch without that check. All three parts matter: a fetch alone is no finding, and the presence of some check does not show it is the ownership check.

Figure 1 shows the GFP from the identifier to the fetch in each version.

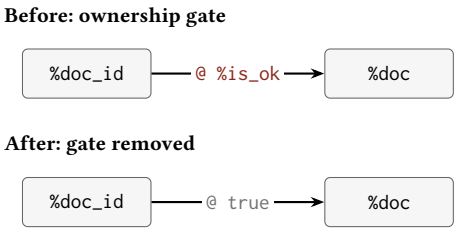


Figure 1: Guarded Flow Paths (GFPs) before and after the patch, with the same source and fetch-result endpoints. `%doc` is the result of `db_fetch_doc`. The edge labels are the actual recovered annotations: `%is_ok` identifies the ownership result being nonzero; it is not a reconstruction of arbitrary predicate semantics.

Comparing the two GFPs reports `%doc_id -> %doc` as newly unconditioned and supplies the second path as its witness path. Read together with the checker, the report explains which input selects the document and which required guard the patch removed.

3 Threat Model and Scope

A checker specifies where attacker-controlled data enters the analysis, which uses are sensitive, and which protections establish the required property. Our function-level experiments assume that the selected inputs are attacker-controlled at function entry; the prototype does not establish that assumption from callers. The intended check concerns the flow from each specified source to its sink, including protection along that flow. Moving a check before the selected source does not protect against attacker control introduced at that source. If the caller instead constrains what can reach the function, those constraints must inform the source specification or be included by extending the analyzed path. The experiments demonstrate the machinery within this boundary, rather than establish end-to-end exploitability for a surrounding application.

4 Guarded Flow Paths

GFP represents value flow as a graph of values, computations, and guarded uses; a path through this graph shows how a value reaches an operation and the guards along that route. The graph has four primitives (Figure 2):

```
G ::= (V, E) -- guarded flow graph
v ::= root(id) | derived(id, op) -- root = (outside frame)
op ::= load | store | cast:* | call(f) | icmp | add | ...
g ::= true | v -- guard: unconditional, or a value-predicate
e ::= v --[g]--> op -- v is USED by op under guard g (a Use)
    | op --> v -- op DEFINES output value v (a Def)

-- Normalized binding forms (each carries its guard g):

   def x <- y @ g (flow)
   def x <- op(y1..yn) @ g (compute)
   def x <- const(k) @ g (const)
```

Figure 2: Guarded value-flow syntax underlying GFP. Guards are ordinary values, so a guard is a graph node and guard removal is an edge whose label changes from a predicate to `true`.

- **Value**: an immutable data item with identity—an SSA variable, a constant, a function name, a stack slot. A value with no in-graph producer is a **Root** (written ⊛): a boundary of what is known—a parameter, global, or allocation whose origin is outside the current frame.
- **Compute**: an operation event consuming values and producing one value; its `op` is an opaque label (e.g. `load`, `cast:inttoptr`, `call(@write)`).
- **Def** / **Use**: the two directions of flow—a compute *defines* its output value; a value is *used* by a compute.
- **Guard**: a condition on a *use*. `None` (`@ true`) means the flow is unconditional; `Some(`g`)` means the flow occurs only when value g holds. Guards are values in the same graph, so "guarded by g" is an edge label and "guard removed" is a label change.

Guarded slice of an operand. Selecting source values S and a sensitive operand T yields the operand's guarded backward slice, $\mathit{BoundedFlow}(S,T) = \mathit{fwd}(S) \cap \mathit{bwd}(T)$: every path by which S reaches T, together with the guards along it. A checker specification states, over this slice, the condition under which T becomes a violation.

Semantics. A *path* $\pi = e_1 e_2 \cdots e_n$ is a sequence of use-edges from a value to a compute and on to that compute's output. Its guard is the conjunction $\mathrm{guard}(\pi) = \bigwedge_i g_i$ of the edge guards along it. A path is **unguarded** iff $\mathrm{guard}(\pi) \equiv$ `true`—every edge is `@ true`. Write $\mathrm{reach}(s,t)$ when some path connects source s to operation t, and $\mathrm{unguarded}(s,t)$ when some *unguarded* such path exists. This is the whole semantics we need: no abstract domain, no fixpoint—reachability over a labelled graph.

Findings are structural. An unguarded path shows that a source reaches an operation with no recorded guard. It does not by itself establish that the path is feasible, that aliases are resolved, or that a guard which is present is the check the operation requires (§9). Which operation is sensitive, and which check it requires, is stated by the checker specification (§6).

5 Differential Authority Analysis

Differential authority analysis runs the same checker on both versions of a function and subtracts. A complete comparison must also say which operations correspond across versions and how a changed guard affects the checker's condition; the prototype implements a narrower, structural comparison.

Let U_v be the source/endpoint pairs joined by an unguarded path (§4) in version v, an endpoint being any value the source reaches (the prototype says *unconditioned*). It reports

$$\Delta_U = U_{\text{after}} \setminus U_{\text{before}}$$

with a witness path for each new pair from the after graph; the vanished guard is visible on the same path in the before graph. The checker says which endpoints are sensitive operations. Whether a present guard is the kind the operation requires (guard adequacy) is the planned second step and needs the checker, not the graph alone.

6 Checker Specifications

A checker specification (Figure 3) lets the analyst state which flows are bad and under what condition: it names source values and sensitive operations, then a condition on the paths connecting them. For example, an untrusted value must not reach a memory dereference without validation. The graph supplies the operations, dependencies, and guards; the checker supplies the condition. A witness path shows how a selected value reaches the operation and which guards occur along the way.

A checker uses `sources` to select values and `sinks` to select operations or call arguments. The `violation` field states the path condition that is a finding; `sanitizers` names operations that may count as the check. Figure 3 excerpts the checker for the integer example of §6.1: a violation is an unconditioned path from a parameter to a selected operation with no comparison involving the source. The analyst chooses these roles and conditions; there is no fixed list of bad operations.

The ownership checker of §2 has the same form: the source is the supplied document identifier, the sensitive operation is the fetch, and the condition is that the identifier reaches it without an ownership check for that caller and document. The integer example below concerns a load whose address comes from a converted integer. Both are one shape, an input reaching a sensitive operation without the check that operation requires, and one query serves both.

6.1 From an integer input to a memory read

Initially, the example function uses an untrusted input as a write count, rejecting values outside 0–255; the supplied buffer is assumed large enough for accepted counts. The patch is constructed for illustration rather than taken from a real commit: it removes those checks and adds the following operations, where `%user_val` is the signed 32-bit input and `%safe_buf` is the buffer:

```
%uval64 = sext i32 %user_val to i64
%danger_ptr = inttoptr i64 %uval64 to i32*
%leaked = load i32, i32* %danger_ptr
%write_size = add i32 %user_val, %leaked
%write_size64 = sext i32 %write_size to i64
%result = call i64 @write(i32 1,
    i8* %safe_buf, i64 %write_size64)
```

Sign extension widens the integer to 64 bits (`%uval64`); `inttoptr` converts it to a pointer; `load` reads through that pointer. This chain

```yaml
name: untrusted-input-deref
sources:
  - kind: param
    all_params: true
sinks:
  - kind: cast
    op: "cast:inttoptr"
  - kind: load
sanitizers:
  - kind: node
    op_prefix: "icmp"
violation:
  path: unconditioned
  sanitizer_absent:
    icmp_for_source: true
```

Figure 3: Excerpt from the written `untrusted-input-deref` checker specification, with comments, the `version`, `description`, `message`, and `severity` fields, and the call-sink selector omitted. Sources, sinks, and a path condition specify the bad flow.

shows how the input selects a memory address to read. The loaded value also contributes to the byte count passed to `write`. The patch therefore introduces an input-derived memory read as well as removing the original bound on the write count.

The checker selects all parameters with `all_params: true`, so `%user_val` is among its sources. Its cast and load selectors identify the operations on the memory-read path. The full checker also includes a `callee_regex` call selector with an argument `index`; for `write`, index 2 selects the byte count. This operand matters: an input influencing the byte count needs a different check from an input selecting the address of a load. A sufficient count bound need not validate a memory address.

Three commands make the finding inspectable. On `after.ll`, `normalize` exposes the operations above, each annotated `@ true`, and shows that the input comparisons are absent; `unconditioned` confirms which endpoints are reachable from `%user_val` along all-unconditional paths. Finally, `diff` on `before.ll` and `after.ll` compares the two versions and returns pairs that have become unconditioned, with witness paths including:

```
%user_val --[true]--> %uval64
        --[true]--> %danger_ptr
        --[true]--> %leaked
```

This witness path gives the complete represented chain from the integer to the loaded value, with no recorded guard on any edge. Reading it with the checker explains the new memory access. `diff` also returns paths to the `write` result through `%write_size64`; inspecting the call operands identifies the unchecked count. In all, it reports nine newly unconditioned source/endpoint pairs and thirteen witness paths on this example (§8); the sixteen other written checkers have the same form.

7 Implementation

AuthDiff lifts LLVM IR [12] to a graph of values, computations, and guarded uses, taking each guard from the branch condition that governs the use, and answers GFP queries over that graph: slicing, the guarded slice, unconditioned reachability, and the comparison of source/endpoint pairs across versions with a witness path for each difference. Figure 4 shows where these sit in the full pipeline; the Check stage, which evaluates a checker specification over the guarded slice, is future work.

The guarded graph could also be built from a program dependence graph [9], with control dependence supplying the guards and slicing [10] selecting the scope; we have not implemented that alternative. What we add to static value-flow analysis [11] is the checker

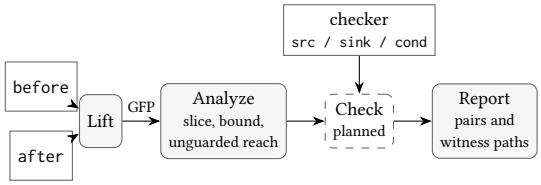


Figure 4: Pipeline of AuthDiff. Two versions are lifted to GFP, analyzed, and compared; Report emits the newly unconditioned pairs and their witness paths. Check (planned) will evaluate a checker specification over the guarded slice.

over the guarded slice and the comparison across versions. The prototype is intra-procedural; §9 describes the planned extension.

8 Evaluation

We ask three questions. (Q1) Does comparing two GFPs report the operations a patch newly exposes, with usable witness paths? (Q2) Across bug classes, what do existing tools provide that GFP does not, and vice versa? (Q3) Do the classes correspond to real vulnerabilities?

Suite. We consider seventeen checker specifications—ten CWE classes (DC-01 – DC-10) and seven OWASP classes (OW-01 – OW-07)—each with a function-level with-check/without-check LLVM-IR example. The tables below mark recorded graph-query results and our assessments. **We are explicit that these examples are synthetic**: hand-written minimal IR models of each class, not real programs, with the single exception of a real excerpt from OpenSSL 1.0.1f for the Heartbleed class. The study asks whether GFP expresses each class as a removed guard and names the authority it exposes; it is not a detection study on real code, and we make no claim of finding real or novel vulnerabilities. Evaluation on real programs and their patch histories is future work (§9). For comparison we consider Infer (Pulse and taint mode), CodeQL, the Clang Static Analyzer, Semgrep, and Coverity; we ran the Clang Static Analyzer and Semgrep on the examples and Infer on the worked example, and assess CodeQL and Coverity from their documented checkers and published coverage. The coverage claims below are therefore a mix of direct runs and documentation-based assessment; that suffices for the qualitative point, what *kind* of result each tool produces, and a full study would make it uniform.

Q1: what the comparison reports. On the integer example (§6.1), comparing the two GFPs reports nine newly unconditioned

Table 1: CWE-class coverage (DC-01 – DC-10). GFP: R denotes a graph-analysis result recorded in the checker suite. Inf.: Infer; CQL: CodeQL; CSA: Clang Static Analyzer; SG: Semgrep; Cov.: Coverity. CSA and SG cells record direct runs on the DC fixtures; Inf., CQL, and Cov. cells are assessed from documented checkers (Inf. was run on the worked example only). • native, ○ partial/configured, "–" not supported.

ID	CWE	GFP	Inf.	CQL	CSA	SG	Cov.
DC-01	476	R	•	•	•	–	•
DC-02	822/843	R	○	•	–	–	○
DC-03	125/787	R	–	•	–	–	•
DC-04	416	R	•	•	•	–	•
DC-05	190/122	R	○	•	–	–	•
DC-06	134	R	–	•	–	–	•
DC-07	415	R	•	•	•	–	•
DC-08	195/196	R	○	○	–	–	•
DC-09	22	R	–	•	–	–	•
DC-10	457	R	○	•	•	–	•

source/endpoint pairs with thirteen witness paths. Inspection connects selected paths to an input-derived address and an unchecked write length. On a Heartbleed-style example, removing the payload-length validation yields new paths including `req` → `sent`, exposing the modeled copy without its length bound.

Q2: what the comparison adds. Q2 asks which tools report the removed check and its path (Table 1). On the worked case, Infer reports nothing on either version, by default and with a Pulse taint model naming `user_val` as a source and the `write` count as a sink; the Clang Static Analyzer reports nothing, and our Semgrep runs, which loaded no rule set, produced no findings. Comparing the two GFPs supplies the newly unconditioned pairs and their witness paths. These observations concern the tested configurations; CodeQL and other frameworks support custom flow and path queries, and we have not established that an equivalent comparison is inexpressible in them. The benefit we claim is that one checker form—sources, a sensitive operation, a condition—yields before/after evidence; its generality and the effort of writing checkers are open (§9).

The OWASP checkers (OW-01 – OW-07) cover SQL and command injection, unauthorized object access, hardcoded credentials, SSRF, insecure deserialization, and authentication bypass. In the access-control classes the exposed authority is plain: an identity or object identifier reaches a privileged operation once its check is changed. Table 2 records our assessment of them and external-tool coverage, pending fuller runs.

Q3: CVE grounding. Each class's example is patterned on a guard that a real, high-impact security patch changed; Table 3 lists representative vulnerabilities. We stress these are *patterns*: the examples are minimal IR models of the guard, not the full programs.

9 Limitations and Future Work

Checker and implementation limits. A checker is only as good as its choice of sources and sensitive operations and of the check each operation requires. A mask may bound a write length if the masked

Table 2: OWASP-class coverage (OW-01 – OW-07), authors' assessment. GFP: A denotes an assessed specification/example. Other tools (Inf.: Infer; CQL: CodeQL; CSA: Clang Static Analyzer; SG: Semgrep; Cov.: Coverity): • native, ○ partial/configured, "–" not supported.

ID	Class (CWE)	GFP	Inf.	CQL	CSA	SG	Cov.
OW-01	SQLi (89)	A	–	•	–	○	•
OW-02	Cmd inj. (78)	A	–	•	–	○	•
OW-03	IDOR (639)	A	–	○	–	–	–
OW-04	Hardcoded (798)	A	–	•	–	•	•
OW-05	SSRF (918)	A	–	•	–	○	○
OW-06	Insec. deser. (502)	A	–	○	–	–	○
OW-07	Auth bypass (306)	A	–	○	–	○	–

Table 3: Representative real vulnerabilities whose patches turn on a guard the corresponding GFP checker models. Examples are minimal models patterned on these classes, not the full programs.

GFP checker class	CVE	Product
Missing bounds check	CVE-2014-0160	OpenSSL (Heartbleed)
Use-after-free	CVE-2019-2215	Android Binder
Command injection	CVE-2014-6271	Bash (Shellshock)
Injection (fmt/JNDI)	CVE-2021-44228	Log4j (Log4Shell)
Path traversal	CVE-2021-41773	Apache httpd
Insecure deser.	CVE-2021-1675	Windows Print Spooler
Authentication bypass	CVE-2021-26855	MS Exchange (ProxyLogon)

value reaches the write and the bound suffices; an operation named and does not establish that. A check removed as redundant leaves an earlier constraint as the protection, seen only if it lies on the path and the checker asks for it. The prototype keeps compute operations and follows edges marked unconditional, but infers no such guarantee. Its guard annotations omit branch polarity and conjunctions of nested conditions. Guard adequacy (§5) and correspondence of paths across versions remain future work. In the integer example, several of the nine newly unconditioned pairs end at intermediate values and two start at the buffer input; these are graph-path witnesses, not triggering inputs and not nine classified bugs.

Interprocedural slices. We plan to place sources at actual input boundaries and extend flows across calls, using LLVM call-graph and pointer-analysis infrastructure for argument, return, and memory relationships. This should let a selected slice include a check in a caller and a sensitive operation in a callee. Summary precision, indirect calls, path conditions, and scalability require evaluation.

A checker language and evaluator. The seventeen checkers are written; a general evaluator that runs them over the guarded slice is future work. Its language must define operand roles, source selection, required guards, and the comparison across versions, and its semantics must separate structural reachability from adequate protection and say how paths correspond across revisions.

Evidence on real patches. Beyond the localized examples, we will evaluate real programs and patch histories, including effective and ineffective masks, relocated checks, and checks made redundant by an earlier constraint. We will separate errors in recovered flows from errors in checker evaluation, and compare equivalent checkers and path reports in existing frameworks. Measuring the effort of writing checkers alongside accuracy and runtime will test the intended benefit.

10 Related Work

Value-flow and taint analyses recover where data goes [11]; program dependence graphs [9] and slicing [10] provide related flow and control views. Infer [4], CodeQL [5], Coverity [6], the Clang Static Analyzer [7], and Semgrep [8] provide bug checkers or configurable analysis frameworks. Automatic patch-based exploit generation [1] demonstrates another use of security-relevant differences. Our checkers organize the comparison around a sensitive operation, the guard it requires, and the path that explains the change. The current study does not establish a general expressive-power advantage over these systems.

CIVSCOPE analyzes data flows through compartment interfaces to characterize potential memory-corruption risks [15]. GFP focuses on the change across a patch: which paths from a source to a sensitive operation become unguarded after a check is removed. Hi-Res derives context-sensitive behavioral fingerprints from object-granular memory traces for exploit detection [16]; GFP instead compares statically recovered guarded flows between program versions.

11 Conclusion

Guarded Flow Paths record, for every value, the condition under which it flows, so a check is part of the graph and a removed check is a difference between two graphs. A checker specification names sources, a sensitive operation, and the condition under which that operation becomes a violation, and one reachability query serves seventeen classes. Run on both sides of a patch, that query reports the operations that lost their guard, with the path and the guard as evidence; what an input now reaches through them is the exposed authority. Our prototype demonstrates lifting, comparison, and witness reporting on localized examples. Interprocedural slices and evaluation on real patch histories are the next steps toward reviewing a patch by the guards it changes [13].

References

[1] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 143–157, 2008.

[2] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[3] Q. Wu and K. Lu. On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits. Accepted to the IEEE Symposium on Security and Privacy (S&P) 2021; withdrawn by the authors before publication. Preprint, 2021.

[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods (NFM)*, LNCS 9058, pages 3–11, 2015.

[5] P. Avgustinov, O. de Moor, M. Peyton Jones, and M. Schäfer. QL: Object-oriented queries on relational data. In *30th European Conf. on Object-Oriented Programming (ECOOP)*, LIPIcs 56, pages 2:1–2:25, 2016.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[7] The Clang Team. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[8] Semgrep, Inc. Semgrep: Lightweight static analysis for many languages. https://semgrep.dev/.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[10] M. Weiser. Program slicing. In *Proc. 5th Int. Conf. on Software Engineering (ICSE)*, pages 439–449, 1981.

[11] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proc. 25th Int. Conf. on Compiler Construction (CC)*, pages 265–266, 2016.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[13] Cybersecurity and Infrastructure Security Agency, Defense Advanced Research Projects Agency, Office of the Under Secretary of Defense for Research and Engineering, and National Security Agency. Closing the software understanding gap. U.S. Government report, 2025. https://www.darpa.mil/news/2025/closing-software-understanding-gap.

[14] Y. Yang, W. Huang, K. Kaoudis, and N. Dautenhahn. Whole-program privilege and compartmentalization analysis with the object-encapsulation model. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 1–12, 2023. https://doi.org/10.1109/SPW59333.2023.00018.

[15] Y. Chien, V.-A. Bădoiu, Y. Yang, Y. Huo, K. Kaoudis, H. Lefeuvre, P. Olivier, and N. Dautenhahn. CIVSCOPE: Analyzing potential memory corruption bugs in compartment interfaces. In *Proc. 1st Workshop on Kernel Isolation, Safety and Verification (KISV)*, pages 33–40, 2023. https://doi.org/10.1145/3625275.3625399.

[16] Z. Yang, S. Solanki, S. Rixner, and N. Dautenhahn. Hi-Res: Precise exploit detection using object-granular memory monitoring. In *2025 IEEE Security and Privacy Workshops (SPW)*, pages 79–90, 2025. https://doi.org/10.1109/SPW67851.2025.00012.

Generative AI Usage

In accordance with the ACM Policy on Authorship, we disclose the use of generative AI tools in preparing this submission. We used a large language model (Anthropic Claude, via an agentic assistant) to draft prose from the authors' prior specifications, experiment reports, and design vocabulary, to format the document in the ACM template, and to assemble an initial reference list. The authors directed the framing and technical content, critically reviewed all AI-assisted text as their own, verified every citation and every reported experimental number against source artifacts, and confirmed that all tool-comparison results reflect actual runs or are explicitly marked as the authors' assessment. The authors retain full responsibility for the accuracy, originality, and integrity of the work.

Supplement: Checker Specifications

Changes in the Guard: Detecting Authority-Exposing Patches with Differential Guarded Flow Paths

(SURE '26)

Nathan Dautenhahn
Dartmouth College and Serenitix
Hanover, NH, USA
Nathan.D.Dautenhahn@dartmouth.edu

Sergey Bratus
Dartmouth College
Hanover, NH, USA
Sergey.L.Bratus@dartmouth.edu

This supplement lists the seventeen checker specifications referenced in the paper (Reviewer A requested them). They are written specifications; the prototype runs graph and differential primitives rather than a general YAML evaluator.

A Checker specifications

These YAML listings preserve the policy fields of seventeen current GFP checker specifications (version 0.1). They are written specifications: the inspected prototype runs graph and differential primitives rather than a general YAML evaluator. Comments and prose `_note` metadata are omitted; selectors and violation fields are unchanged. An omitted sanitizer field is not synthesized.

A.1 DC-01: Null dereference

```yaml
name: null-deref
version: '0.1'
sources:
  - kind: const
    value: 'null'
  - kind: param
    pointer: true
  - kind: call
    callee_regex: "^(malloc|calloc|realloc|strdup|fopen)$"
sinks:
  - kind: load
  - kind: store
violation:
  path: unconditioned
  structural_guard_absent:
    icmp_for_operand: source
severity: error
```

A.2 DC-02: Untrusted-input dereference

```yaml
name: untrusted-input-deref
version: '0.1'
sources:
  - kind: param
    all_params: true
sinks:
  - kind: cast
    op: cast:inttoptr
  - kind: call
    callee_regex: "^(@write|@send|@fwrite|@memcpy|@memmove)$"
    index: 2
  - kind: load
sanitizers:
  - kind: node
    op_prefix: icmp
violation:
  path: unconditioned
  sanitizer_absent:
    icmp_for_source: true
severity: error
```

A.3 DC-03: Missing bounds check

```yaml
name: missing-bounds-check
version: '0.1'
sources:
  - kind: param
    all_params: true
  - kind: call
    callee_regex: "^(@recv|@read|@recvfrom|@recvmsg)$"
  - kind: load
sinks:
  - kind: call
    callee_regex: "^(@llvm\\.memcpy\\..*|@memcpy|@memmove|@bcopy)$
  "
    index: 2
  - kind: call
    callee_regex: "^(@write|@send|@sendto|@fwrite)$"
    index: 2
sanitizers:
  - kind: node
    op_prefix: icmp
violation:
  path: unconditioned
  sanitizer_absent:
    icmp_for_source: true
severity: critical
```

A.4 DC-04: Use after free

```
name: use-after-free
version: '0.1'
sources:
  - kind: param
    pointer: true
  - kind: call
    callee_regex: "^(@malloc|@calloc|@realloc|@strdup|@strndup)$"
sinks:
  - kind: call
    callee_regex: "^@free$"
    index: 0
  - kind: load
  - kind: store
sanitizers:
  - kind: store
    value: 'null'
violation:
  path: unconditioned
  co_occurrence:
    all_of:
    - sink: call(@free, ptr)
    - sink: load(ptr) | store(val, ptr)
  sanitizer_absent:
    null_store_for_ptr: true
severity: critical
```

A.5 DC-05: Integer overflow to allocation

```
name: int-overflow-alloc
version: '0.1'
sources:
  - kind: param
    all_params: true
sinks:
  - kind: call
    callee_regex: "^(@malloc|@calloc|@realloc|@aligned_alloc|
  @kmalloc|@vmalloc)$"
    index: 0
sanitizers:
  - kind: node
    op_prefix: icmp
  - kind: call
    callee: "@__builtin_mul_overflow"
violation:
  path: unconditioned
  intermediate_required: mul
  sanitizer_absent:
    icmp_for_source: true
severity: critical
```

A.6 DC-06: Format string

```
name: format-string
version: '0.1'
sources:
  - kind: param
    all_params: true
  - kind: call
    callee_regex: "^(@read|@fgets|@gets|@getenv|@recv)$"
sinks:
  - kind: call
    callee_regex: "^(@printf|@fprintf|@sprintf|@snprintf|@vprintf|
  @syslog|@vsyslog)$"
    index: 0
sanitizers:
  - kind: root
    global: true
violation:
  path: unconditioned
  sanitizer_absent:
    global_literal_in_call_inputs: true
severity: critical
```

A.7 DC-07: Double free

```
name: double-free
version: '0.1'
sources:
  - kind: param
    pointer: true
  - kind: call
    callee_regex: "^(@malloc|@calloc|@realloc|@strdup|@strndup)$"
sinks:
  - kind: call
    callee_regex: "^@free$"
    index: 0
sanitizers:
  - kind: store
    value: 'null'
  - kind: call
    callee_regex: "^(@malloc|@calloc|@realloc)$"
violation:
  path: unconditioned
  sink_count: ">= 2"
  sanitizer_absent:
    null_store_between_frees: true
severity: critical
```

A.8 DC-08: Signedness

```
name: signedness
version: '0.1'
sources:
  - kind: param
sinks:
  - kind: call
    callee_regex: "^(@malloc|@calloc|@realloc|@kmalloc|@vmalloc|
  @kzalloc)$"
    index: 0
  - kind: gep
sanitizers:
  - kind: node
    op_prefix: icmp
    icmp_guard: slt 1 | sle 0 | sgt 0 | ult 1
violation:
  path: unconditioned
  intermediate_required: cast:sext
  sanitizer_absent:
    icmp_slt_for_source: true
severity: high
```

A.9 DC-09: Path traversal

```yaml
name: path-traversal
version: '0.1'
sources:
  - kind: param
    all_params: true
  - kind: call
    callee_regex: "^(@getenv|@gets|@fgets|@recv|@read)$"
sinks:
  - kind: call
    callee_regex: "^(@open|@openat|@creat|@fopen|@freopen)$"
    index: 0
  - kind: call
    callee_regex: "^(@execve|@execvp|@system|@popen)$"
    index: 0
sanitizers:
  - kind: call
    callee_regex: "^(@realpath|@canonicalize_file_name)$"
  - kind: node
    op_prefix: icmp
violation:
  path: unconditioned
  sanitizer_absent:
    no_realpath_on_path: true
severity: high
```

A.10 DC-10: Uninitialized read

```yaml
name: uninit-read
version: '0.1'
sources:
  - kind: alloca
sinks:
  - kind: load
sanitizers:
  - kind: store
    target_is_source: true
violation:
  path: unconditioned
  sanitizer_absent:
    store_for_alloca: true
severity: high
```

A.11 OW-01: SQL injection

```yaml
name: sql-injection
version: '0.1'
sources:
  - kind: param
    pointer: true
sinks:
  - kind: call
    callee_regex: "^(sqlite3_exec|sqlite3_prepare|PQexec|
  PQexecParams|mysql_query|db_query|exec_sql|format_sql)$"
sanitizers:
  - kind: call
    callee_regex: "^(db_escape|sqlite3_bind|PQescapeLiteral|
  mysql_real_escape_string|db_prepare)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: critical
```

A.12 OW-02: Command injection

```yaml
name: cmd-injection
version: '0.1'
sources:
  - kind: param
    pointer: true
sinks:
  - kind: call
    callee_regex: "^(system|popen|execvp|execve|execl|execlp|
  execvpe|build_ping_cmd|run_cmd|shell_exec)$"
sanitizers:
  - kind: call
    callee_regex: "^(validate_hostname|validate_path|validate_arg|
  sanitize_shell|escape_shell)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: critical
```

A.13 OW-03: Broken access control

```yaml
name: broken-access-control
version: '0.1'
sources:
  - kind: param
    name_regex: ".*(id|Id|ID|key|Key).*"
sinks:
  - kind: call
    callee_regex: "^(db_fetch_doc|db_fetch_record|db_get|
  fetch_resource|read_file_for_user)$"
sanitizers:
  - kind: call
    callee_regex: "^(acl_check|check_ownership|authorize|
  permission_check|is_owner)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: critical
```

A.14 OW-04: Hardcoded credentials

```yaml
name: hardcoded-credentials
version: '0.1'
sources:
  - kind: global_const
    name_regex: ".*(key|Key|KEY|secret|Secret|SECRET|iv|IV|passwd|
  password|salt).*"
sinks:
  - kind: call
    callee_regex: "^(aes_encrypt|aes_decrypt|hmac_init|
  EVP_EncryptInit|EVP_DecryptInit|crypto_secretbox|
  ssl_set_psk_server_callback|aes_cbc_encrypt)$"
sanitizers:
  - kind: call
    callee_regex: "^(get_key_from_config|get_key_from_env|
  PKCS5_PBKDF2_HMAC|EVP_BytesToKey|derive_key|read_keyfile)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: high
```

A.15 OW-05: SSRF

```
name: ssrf
version: '0.1'
sources:
  - kind: param
    pointer: true
sinks:
  - kind: call
    callee_regex: "^(http_get|http_post|curl_easy_setopt|
  curl_easy_perform|tcp_connect|dial_tcp|fetch_url)$"
sanitizers:
  - kind: call
    callee_regex: "^(validate_url_allowlist|parse_and_check_host|
  url_is_internal|check_url_allowlist)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: high
```

A.16 OW-06: Insecure deserialization

```
name: insecure-deserialization
version: '0.1'
sources:
  - kind: param
    pointer: true
sinks:
  - kind: cast
    op: inttoptr
  - kind: call
    indirect: true
sanitizers:
  - kind: call
    callee_regex: "^(validate_type|type_is_known|dispatch_by_type|
  check_type_bounds)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: critical
```

A.17 OW-07: Authentication bypass

```
name: auth-bypass
version: '0.1'
sources:
  - kind: param
    name_regex: ".*(user|User|USER|token|Token|TOKEN|session|
  Session|id|Id|ID).*"
sinks:
  - kind: call
    callee_regex: "^(fetch_admin_records|write_admin_config|
  delete_record|fetch_private_data|get_all_users|reset_password)$"
sanitizers:
  - kind: call
    callee_regex: "^(authenticate|authorize|verify_token|
  verify_session|check_admin|require_auth|is_authenticated)$"
violation:
  path: unconditioned
  sanitizer_absent: true
severity: critical
```