International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-024-00764-z

GENERAL

When long jumps fall short: control-flow tracking and misuse detection for nonlocal jumps in C

Extended version

Julian Erhard[1,2] · Michael Schwarz[1] · Vesal Vojdani[3] · Simmo Saan[3] · Helmut Seidl[1]

Accepted: 14 August 2024

Abstract
The C programming language offers `setjmp/longjmp` as a mechanism for nonlocal control flow. This mechanism has complicated semantics. As most developers do not encounter it day-to-day, they may be unfamiliar with all its intricacies – leading to subtle programming errors. At the same time, most static analyzers lack proper support, implying that otherwise sound tools miss whole classes of program deficiencies. We propose a concrete semantics of a subset of C with `setjmp/longjmp`, where interprocedural `longjmps` are performed directly, as well as an equivalent formulation where such jumps are implemented via stack-unwinding at the call-sites. Reflecting this semantic equivalence, we propose an approach for lifting existing interprocedural analyses to support `setjmp/longjmp` and to flag their misuse. To deal with the nonlocal semantics, our approach leverages side-effecting transfer functions, which, when executed, may additionally trigger contributions for program points that are not static control-flow successors. We showcase our analysis on a real-world example and propose a set of litmus tests for other analyzers.

Keywords Abstract interpretation · Static analysis · `setjmp/longjmp` · Side-effecting constraint systems

1 Introduction

For statically analyzing real-world programs, analysis developers are confronted with a wealth of intricate language features. Therefore analyzers often focus on a subset of the programming language, ignoring some more obscure features and making optimistic assumptions about others. As noted by Livshits et al. [13], this is true even for tools that claim to be *sound*, i.e., not to miss any bug. The authors provide a checklist for static analyzers, making it easier for analysis authors to indicate which features are supported and which are not. For the C programming language, they mention `setjmp/longjmp` as an example of often unsupported language features. The functions `setjmp/longjmp` allow defining exceptional control flow by dynamically jumping up the callstack. Indeed, these nonlocal jumps are not supported even in many state-of-the-art tools [2, 4, 12, 17, 21]. Empirical studies by Christakis and Bird [6] indicate that developers think it of exceptional importance to cover exceptional control flow. Therefore the lack of support for this C language feature is unsatisfactory.

The `setjmp/longjmp` mechanism allows saving the current state of execution into a *jump buffer* by means of `setjmp`. At a later point of program execution, parts of the callstack may be abandoned by calling `longjmp`. Execution then continues at the stackframe specified by the jump buffer. This is not only conceptually intricate, but fraught with many caveats. For example, accessing the values of nonvolatile locals that have been modified between the call to `setjmp` and the call to `longjmp` is Undefined Behavior. As developers typically only use `setjmp/longjmp` in few selected loca-

✉ J. Erhard
julian.erhard@tum.de

M. Schwarz
m.schwarz@tum.de

V. Vojdani
vesal.vojdani@ut.ee

S. Saan
simmo.saan@ut.ee

H. Seidl
helmut.seidl@tum.de

1 TUM School of Computation, Information and Technology, Technical University of Munich, Garching, Germany

2 Institute of Informatics, Ludwig-Maximilians-Universität in Munich, Munich, Germany

3 Institute of Computer Science, University of Tartu, Tartu, Estonia

Published online: 29 August 2024

```
 1  jmp_buf_t errorhandler;
 2  int error;
 3  void foo() {
 4    // ...
 5    int z = get_status();
 6    if(z < 0) {
 7      error = 42;
 8      longjmp(errorhandler, error);
 9    }
10    // ...
11  }
12  void bar() {
13    char* logpath;
14    if(setjmp(errorhandler)) {
15      printf("error encountered!");
16      // Bug!
17      write_log(logpath, error);
18      exit(1);
19    }
20    logpath = get_logpath();
21    foo();
22  }
23  void main() {
24    bar();
25    // ...
26    int z = get_status();
27    if(z < 0) {
28      error = -17;
29      // Bug!
30      longjmp(errorhandler, error);
31    }
32  }
```

Fig. 1 Program fragment making use of `setjmp/longjmp` that contains two bugs

tions, familiarity with these intricacies is not widely spread, resulting in potential vulnerabilities [14, 15]. Therefore a static analyzer should not ignore `setjmp/longjmp`. Instead, not to fall short of user expectations, it should take this feature into account during its analysis of other programming deficiencies, *as well as* warn about potential misuses. While exception handlers in Java or C++ are well structured, this is not necessarily the case for `setjmp/longjmp`, where the jump target may depend on the jump buffer's runtime value, making the analysis of `setjmp/longjmp` challenging.

We took up this challenge and demonstrate how existing analyses for C can be lifted to support `setjmp/longjmp` by reusing existing building blocks of interprocedural analyses, instead of dedicated mechanisms. We propose that the analysis performs an *abstract stack unwinding* complemented with

- an analysis of currently valid jump targets;
- a value analysis for jump buffers, and
- taint analysis to check for illegal accesses to locals.

For stack unwinding and the collection of abstract states at `setjmp` locations, we rely on *side effects* in transfer functions. Side-effecting constraint systems [1, 25] allow accumulating flow-insensitive information during a flow- and context-sensitive analysis. They have been used, e.g., for the analysis of the values of global variables [22, 24], expressing a variety of approaches to context-sensitivity [1, 8], and for tracking accesses to globals to check for races and invalid dereferences [20, 27]. Here, side effects are used to handle `longjmps` without polluting control-flow graphs with an excessive number of additional edges (e.g., from every procedure call to every invocation of `setjmp`).

The rest of the paper is structured as follows. In Sect. 2, we recall the semantics of `setjmp/longjmp` along an example and identify possible programming errors. Compared to the workshop version of this paper [23], Sects. 3 to 6 are added. Section 3 identifies a core C language with `setjmp/longjmp` and describes an intuitive semantics where `longjmps` are performed directly. Section 4 provides an adapted formulation of the semantics where `longjmps` between function boundaries are propagated via call sites, and the following Sect. 5 establishes an equivalence relationship between the two semantics. In Sect. 6 the semantics are adapted to not continue traces that would contain illegal accesses to indeterminate variables. Turning to the analysis, Sect. 7 describes a generic base approach to interprocedural analysis, which is extended in Sect. 8 to an analysis of `setjmp/longjmp`, where it is argued when an analysis is correct with respect to the concrete semantics from Sect. 4. Section 9 explains how illegal accesses to locals are identified. Section 10 reports on our implementation within an analyzer for multithreaded C based on Abstract Interpretation [7] and the results of a preliminary experimental evaluation. Finally, Sect. 11 discusses related work, and Sect. 12 concludes.

2 `Setjmp/Longjmp` in C

The usage of `setjmp/longjmp` is best explained by an example. The program in Fig. 1 has two global variables, `errorhandler` of the predefined type `jmp_buf` and `error` of type `int`. The `main` function first calls `bar`, which in turn calls `setjmp(errorhandler)`. That call saves the current execution state into the jump buffer `errorhandler`, returning the value `0`. Thus the `if` branch in line 14 is not taken. The function `bar` then sets its local variable `logpath` to the path returned by a call to the external function `get_logpath()`. The program is meant to later record error messages into the file identified by `logpath`. The function `bar` continues with calling `foo`. Inside `foo` at line 5, the status information, as returned by some function `get_status()`, is checked. A negative status is interpreted as an error indication, in which case `error` is set to `42`, and `longjmp(errorhandler, err)` is invoked. This call transfers control back to the invocation of `setjmp` inside of `bar` at line 14 – which now returns with the value `42` passed as the argument to `longjmp`. Now the `if` branch is taken, a message is printed to `stdout`, an error message is written into the file, and the program terminates.

If no error occurs during `foo` on the other hand, `foo` and `bar` both return regularly. Subsequently, the function `main` checks the status again, and if it is negative, then `error` is set to -17, and a call to longjmp(errorhandler, err) occurs. This, though, constitutes a fault in the program, as it performs a longjmp to an invocation of setjmp inside a function that already has returned, resulting in Undefined Behavior.

However, this is not the only fault in the fragment: Consider the case where a longjmp happens from foo at line 8. Here the function bar containing the call to setjmp has *not* returned yet. The issue is different: the local variable logpath has been modified between the calls to setjmp and longjmp. Thus it has Indeterminate Value after the longjmp, resulting in Undefined Behavior at the access in line 17. There is an easy fix, though: Declaring logpath to be volatile ensures that the correct value is read.

The example highlights pitfalls of using setjmp/longjmp, which a static analyzer should be expected to flag. The following misuses are possible and lead to Undefined Behavior:

(A) Calling longjmp on a jmp_buf for which setjmp has not been called.
(B) Calling longjmp when the function containing the corresponding call to setjmp has already returned.
(C) Calling longjmp from a thread different from the one calling setjmp.
(D) Reading a nonvolatile variable x of automatic storage duration after a longjmp where x has been modified between setjmp and longjmp and has not been overwritten since the longjmp.
(E) Calling longjmp on a jmp_buf that has not been initialized via setjmp but instead by copying the content of a different jmp_buf.
(F) Calling longjmp where the corresponding setjmp was within the scope of a variable-length array, and this scope has since been left.

Additionally,

(G) if argument 0 is passed to longjmp, then this argument is silently changed to 1. Although this is not Undefined Behavior, it still likely is a bug, and a warning should be produced.

Furthermore,

(H) any code after a call to longjmp is unreachable, for which an appropriate warning should also be issued. We will discuss how to soundly detect all these possible misuses and issue warnings in Sect. 8.

3 Language and semantics

We for now concentrate on the core subset of the C language, which abandons dynamic memory allocation, recursion, dynamic function calls, and structured data, which are largely orthogonal features. We make this restriction only for clarity of presentation; the implementation of the analysis targets real-world programs and thus supports all these features. Thus a program consists of a finite set of functions $\mathcal{F}$ with $main \in \mathcal{F}$ being the dedicated start function and with setjmp, longjmp $\notin \mathcal{F}$, as well as a set of global variables G. Each function $f \in \mathcal{F}$ is defined by its control-flow graph (CFG), a list of parameters, and a finite set of local variables L_f. The union of all local and global variables is denoted by $\mathcal{V}$. The nodes or program points of the CFG for a function f are from a finite set $\mathcal{N}_f$, with st_f and ret_f referring to the (unique) start and end points of f. For simplicity, we assume that the function's return value is assigned to a dedicated global program variable retv. Each local and global variable has one of the types int, int*, jmp_buf, or jmp_buf*. Variables may be marked with the volatile type modifier. We assume that programs are well-typed.

Let $\mathcal{N}$ denote the disjoint union of the sets $\mathcal{N}_f, f \in \mathcal{F}$. An edge $(u, a, v) \in \mathcal{E}$ of the CFGs proceeds from source node u to sink node v and is labeled with some action a, which may be an assignment, a function call, a guard, function return, or a call to setjmp/longjmp. For a node u, there may only be two edges $(u, a, v), (u,' a, v') \in \mathcal{E}$ with $v \neq v'$ in case that a is a guard and a' is its negated guard. Otherwise, u may only have one successor in $\mathcal{E}$. Assignments in a function f have the form l = e, where l is a variable x or dereference of a variable $*x$ for $x \in G \cup L_f$, and e is some expression of suitable type without side effects or function calls. One particular form of expression is $\&x$ for some variable x, which returns a pointer to x. A guard is given by some side-effect-free expression g of type int. Function calls have the form $\texttt{x = f(a}_1\texttt{,...,a}_n\texttt{)}$ with $f \in \mathcal{F}$, the arguments having suitable types for f, and, for simplicity of exposition, the variable x being of type int. A call to setjmp is represented by control-flow edges as well. According to the C standard, it appears as if a call to setjmp may return more than once: First, when setjmp is invoked, and possibly again whenever a longjmp back to this program point occurred. Accordingly, we introduce *two* control-flow edges for each setjmp: one edge $(u, \texttt{x = setjmp}_0\texttt{(b)}, v)$, which represents the first execution of the call, and an additional edge $(\bar{u}, \texttt{x = setjmp}_1\texttt{(b)}, v)$ for all subsequent returns, i.e., longjmps to this location. The additional edge has the same sink node v, but a fresh source node $\bar{u}$ without predecessors in the CFG[1] (see Fig. 2). Let $\mathcal{B} \subset \mathcal{N}$ denote the set of the barred nodes $\bar{u}$. A call to longjmp is represented by an edge with a label $\texttt{longjmp}(b, a)$, where b is of type jump_buf, and a is of type int. For the treatment of longjmps, we introduce for each function f an artificial extra return node $\mathsf{ret}'_f \in \mathcal{N}$ for irregular returns that has no ingoing or outgoing edges. For the sake of simplicity, direct and indirect

[1] We allow assignments of setjmp to return values. In C, this is not allowed, but one may directly branch over the returned value.

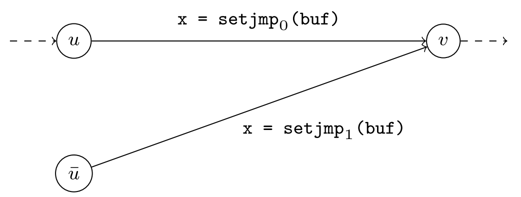


Fig. 2 Representation of a `setjmp` in the control-flow graph using two edges. Returns from `longjmp`s arrive at $\bar{u}$

recursion is not allowed, i.e., no function may have multiple active stack frames at a time. Moreover, we assume that all functions have distinct sets of local variables.

As the concrete semantics of the language, we choose a *trace* semantics. In our setting, a program execution trace $t \in \mathcal{T}$ consists of a sequence of program configurations $(c_i)_{0\leq i\leq k}$ for some $k \in \mathbb{N}$, where each configuration $c_i = (u, \sigma)$ consists of a program node u and a program state $\sigma \in \Sigma$. The program state is a type-correct mapping from program variables to values. The set of initial traces is given by the set init consisting of single-configuration traces c_0 of the form $(\mathsf{st}_{\mathsf{main}}, \sigma)$ with σ providing initial values for global variables and for the locals of main. We formalize the concrete semantics using a constraint system over sets of traces such that the least solution of that system describes all possible program executions. The constraint system has *unknowns* $[u, c]$, $u \in \mathcal{N}$, $c \in \mathcal{T}$, where the value of $[u, c]$ is meant to collect the set of traces reaching program point u when the function containing u is reached with trace c. In particular, the last step of c contains the start node of the procedure and its start state. We call c the (concrete) *context* of $[u, c]$. We rely on *side-effecting constraint systems* for the formulation of the concrete constraint system [1]. Here side-effecting constraint systems allow us to conveniently formalize the semantics of `longjmp`s: Side effects are used to propagate the traces with which one may return to the `setjmp` location. This allows handling jumps across function boundaries at *dynamic jump targets* as required by a formalization of the semantics of calls to `setjmp` and `longjmp`.

Example 1
Let us recall the notion of a side-effecting constraint system along a self-contained example. Consider for now a different set of unknowns given by $\mathbb{U} = \{x, y, z\}$. Each unknown takes a value from the powerset lattice $\mathbb{L} = 2^{\{a,b,c\}}$ ordered by $\subseteq$ with $\bot = \emptyset$ and $\top = \{a, b, c\}$. Assume that we have the following constraint system over $\eta : \mathbb{U} \to \mathbb{L}$:

$$\begin{array}{lcl@{\qquad}lcl}
(\eta, \eta\, x) & \supseteq & f_x\, \eta & f_x\, \eta & = & (\emptyset, \{a\} \cup (\{c\} \cap \eta\, z)) \\
(\eta, \eta\, y) & \supseteq & f_y\, \eta & f_y\, \eta & = & (\{z \mapsto \eta\, y\}, \{b\} \sqcup \eta\, x) \\
(\eta, \eta\, z) & \supseteq & f_z\, \eta & f_z\, \eta & = & (\emptyset, \bot)
\end{array}$$

where the ordering $\supseteq$ is lifted to maps and tuples pointwise. The right-hand sides of the constraints yield both a contribution from $\mathbb{L}$ to the unknown on the left-hand side (second component), as well as a (partial) mapping from unknowns to elements of $\mathbb{L}$ (the *side-effects*) as the first component. Here the right-hand sides are defined using functions $f_i : (\mathbb{U} \to \mathbb{L}) \to (\mathbb{U} \to \mathbb{L}) \times \mathbb{L}$ for $i \in \{x, y, z\}$. In this example, f_y causes a side effect to the unknown z, which does not receive any non-$\bot$ values from other constraints, and contributes $\{b\} \sqcup \eta\, x$ to its left-hand side, i.e., y. A total mapping from unknowns to $\mathbb{L}$ is a solution of the constraint system if it satisfies all constraints. Consider $\eta_1 = \{x \mapsto \top, y \mapsto \top, z \mapsto \top\}$. While η_1 is a solution of the side-effecting constraint system, it is not the *least* solution given by $\{x \mapsto \{a\}, y \mapsto \{a, b\}, z \mapsto \{a, b\}\}$.

The constraint system C for the concrete semantics is given by

$$\begin{array}{ll}
\eta\, [\mathsf{st}_{\mathsf{main}}, t] & \supseteq \{t\},\;\; t \in \mathsf{init}; \\
(\eta, \eta\, [v, c]) & \supseteq [\![(u, a, v), c]\!]\, \eta,\;\; (u, a, v) \in \mathcal{E}, c \in \mathcal{T}.
\end{array}$$

Here $\supseteq$ is lifted to maps and tuples, and $\eta : (\mathcal{N} \times \mathcal{T}) \to 2^{\mathcal{T}}$ is some mapping from unknowns to sets of traces. Thus $\eta\, [u, c]$ represents the set of traces reaching program point u when the function containing u was reached with trace c. The first constraint initializes the unknowns for the start point. The constraint for each edge (u, a, v) and context c ensures that the mapping η accounts for all side effects that occur for the edge (u, a, v) in the context c and that $\eta\, [v, c]$ contains the traces reaching v in context c via that edge. We rely on the following helper functions to define the semantics $[\![(u, a, v), c]\!]$ for each action a: The function $\mathsf{last}\, t$ retrieves the last configuration of the given trace t. As there are no recursive calls, there is at most one invocation per function in t that has not returned yet (either regularly or via `longjmp`). Let $\mathsf{calls}\, t$ denote this set of functions that have not returned. Accordingly, the state in configuration $\mathsf{last}\, t$ consists of a mapping of globals and the locals of all functions $f \in \mathsf{calls}\, t$ to values. The function $\mathsf{func}\, u$ determines for each node u the function it occurs in. The predicate $\mathsf{here}_{u,c}\langle u', t'\rangle$ is given by $(\mathsf{func}\, u = \mathsf{func}\, u') \wedge (c = \mathsf{context}\, t')$, where the function $\mathsf{context}\, t'$ yields the concrete context of the function invocation that is active in the last configuration of t'. In the following, if some expression cannot be evaluated, e.g., a pointer is dereferenced that evaluates to a local variable that is not on the stack, we assume that the corresponding traces will not be constructed and thus not propagated. Let $e = (u, a, v) \in \mathcal{E}$ denote an edge in one of the control-flow graphs, and let c denote some concrete context. Then we define the semantics $[\![e, c]\!]$ by case distinction on the action a. We will use some OCaml-like pseudocode to do so.

Assignment If a is an assignment $l = r$, then

$$\begin{array}{l}[\![e,c]\!]\,\eta = \textbf{let}\; T = \eta\,[u,c] \;\textbf{in}\\ \textbf{let}\; T' = \{t \boxplus (v, \sigma \oplus \{x \mapsto w\}) \mid\\ \quad t \in T, \mathsf{last}(t) = (u,\sigma), \&x = [\![\&l]\!]\,\sigma,\\ \quad w = \textbf{match}\;\mathsf{type\_of}\,r\;\textbf{with}\;\mathsf{jump\_buf} \mapsto \mathsf{err}\\ \qquad\quad \mid \_ \mapsto [\![r]\!]\,\sigma\}\\ \textbf{in}\\ (\emptyset, T')\end{array}$$

Here the operator $\boxplus$ appends a configuration to a trace, and the operator $\oplus$ replaces an entry in a map. Each trace t that reaches the unknown $[u,c]$, with a last state σ is extended with one configuration where the assignment was performed. The variable x, which is assigned to, is determined by the address $\&x$ to which $\&l$ evaluates in σ. To determine the value w to be assigned, the value of the right-hand side expression r in state σ is determined; in case r is of type jump_buf, the value w is set to err, as jump buffers may only be set via `setjmp`. Thus each trace is extended with a configuration consisting of the next program point and the updated state.

`setjmp` A jump buffer value is either an error value err or a pair $\langle \bar{u}, t\rangle$, where $\bar{u}$ is the program point later to be long-jumped to, and t is the trace with which the corresponding `setjmp` was reached. We assume that all variables of type `jmp_buf` are initialized with err. If the action a is `x = setjmp`$_0$`(b)` for some variable x and jump buffer expression b, then

$$\begin{array}{l}[\![e,c]\!]\,\eta = \textbf{let}\; T = \eta\,[u,c] \;\textbf{in}\\ \textbf{let}\; T' = \{t \boxplus (v, \sigma \oplus \{x \mapsto 0,\\ \quad [\![\&b]\!]\sigma \mapsto \langle \bar{u}, t\rangle\}) \mid t \in T, \mathsf{last}(t) = (u,\sigma)\}\\ \textbf{in}\\ (\emptyset, T')\end{array}$$

Each trace is extended with a configuration where in the updated state, x is mapped to 0, and the value of the jump buffer b is set to the jump buffer value $\langle \bar{u}, t\rangle$, where $\bar{u}$ is the source node for all subsequent returns of the `setjmp` operation with source node u, and t is the trace that the `setjmp` operation was reached with.

Now consider an action a of the form `x= setjmp`$_1$`(b)`. In this case, the start point u of the edge is of the form $\bar{u}'$ for some program point u' and an outgoing edge with a corresponding action `x = setjmp`$_0$`(b)`. Here the variable x receives the value from the global variable `retv`, which, as we will see, contains the value set by the immediately preceding `longjmp`. Thus the constraint is the same as for the assignment `x = retv`.

`longjmp` Subsequently, we require a predicate $\mathsf{valid}(b,t)$, which states whether the jump buffer expression b is *valid* for the trace t, i.e., contains a jump buffer value, which may serve as the target of a `longjmp` in the last configuration of t. This predicate is defined by

$$\begin{array}{l}\mathsf{valid}(b,t) = \textbf{let}\;(\_,\sigma) = \mathsf{last}(t)\;\textbf{in}\\ \quad [\![b]\!]\,\sigma \neq \mathsf{err} \wedge\\ \qquad \textbf{let}\;\langle \bar{u}', t'\rangle = [\![b]\!]\,\sigma \;\textbf{in}\\ \qquad \mathsf{onstack}(t', t)\end{array}$$

where it is first checked that the value of the jump buffer expression b is different from err. The value $\langle \bar{u}', t'\rangle$ of b is then obtained, and it is verified with $\mathsf{onstack}(t',t)$ that the function invocation containing the `setjmp` that was reached with t' is still on the call stack in t. Now assume that the action a is `longjmp`(b,x). Then

$$\begin{array}{l}[\![e,c]\!]\,\eta = \textbf{let}\; V = \{t \mid t \in \eta\,[u,c], \mathsf{valid}(b,t)\}\;\textbf{in}\\ \textbf{let}\; H = \{([\bar{u}',c], (r \boxplus (\bar{u}', \sigma'))) \mid\\ \quad r \in V, \mathsf{last}(r) = (u,\sigma),\\ \quad \sigma' = \sigma \oplus \{\mathtt{retv} \mapsto [\![(x == 0)\;?\;1 : x]\!]\,\sigma\},\\ \quad \langle \bar{u}', t'\rangle = [\![b]\!]\,\sigma, \mathsf{here}_{u,c}\langle \bar{u}', t'\rangle\}\\ \textbf{in}\\ \textbf{let}\; O = \{([\bar{u}',c'], r \boxplus (\mathsf{ret}'_{\mathsf{func}\,u}, \sigma') \boxplus (\bar{u}', \sigma'')) \mid\\ \quad r \in V, \mathsf{last}(r) = (u,\sigma),\\ \quad \sigma' = \sigma \oplus \{\mathtt{retv} \mapsto [\![(x == 0)\;?\;1 : x]\!]\,\sigma\},\\ \quad \langle \bar{u}', t'\rangle = [\![b]\!]\,\sigma,\\ \quad \sigma'' = \mathsf{remove\_locals}(\sigma', t'),\\ \quad c' = \mathsf{context}(t'), \neg\mathsf{here}_{u,c}\langle \bar{u}', t'\rangle\}\\ \textbf{in}\\ (\overrightarrow{H \cup O}, \emptyset)\end{array}$$

For a set R of pairs (X,t) of unknowns X and traces t, we denote by $\overrightarrow{R}$ the function defined by

$$\overrightarrow{R}\,X = \{t \mid (X,t) \in R\}.$$

Given a set of pairs, it defines a function that, given an argument X, yields a set containing all t such that $(X,t) \in R$, thus grouping all side effects per unknown. First, the set V of traces r reaching program node u in the context c are determined for which b is valid. Only these traces will be extended and propagated.

The relation H collects pairs of unknowns $[\bar{u}', c]$ and extensions of traces $r \in V$ for which the jump target value bv of b was set by a `setjmp` inside the current function invocation with context c. Each such extended trace is obtained by appending one step where the value of `retv` is updated with the value provided by x. According to the semantics of C, we enforce that this value is necessarily different from 0.

The relation O, on the other hand, collects the corresponding set of pairs of unknowns $[\bar{u}', c']$ and extensions of traces $r \in V$ for which the jump target value $\langle \bar{u}', t'\rangle$ of b was set by a `setjmp` *outside* the current function invocation with context c. In contrast to the traces in H, each trace t from V that jumps outside the current function invocation is extended by *two* steps: the first step takes care of the assignment of the value of x to `retv` resulting in some state σ'. To signify the irregular exit of the current function $\mathsf{func}\, u$, i.e., via an interprocedural `longjmp`, this state σ' is associated with an artificial extra return node $\mathsf{ret}'_{\mathsf{func}\,u}$. We remark that while such nodes may appear in traces, there are no corresponding unknowns in this constraint system. The second step then removes from σ' all local variables from functions that are no longer on the call stack after the `longjmp` is performed. For this second step, let $bv = \langle \bar{u}, t' \rangle$ denote the jump buffer value. The function $\mathsf{remove\_locals}(\sigma', t')$ removes any bindings of local variables from σ' that are not locals of a function from $\mathsf{calls}(t')$, i.e., the set of functions on the stack when the jump buffer value bv was created.

The contributions as specified by H and O are then side-effected, whereas the empty set of traces is contributed to $[v, c]$, indicating that the successor program point cannot be reached via this edge.

Function calls If the action a has the form `x = f(a`$_1$`, ..., a`$_k$`)`, then

$$\begin{aligned}
&[\![e, c]\!]\, \eta = \textbf{let}\ S = \{([\mathsf{st}_f, t'], t') \mid \\
&\quad t \in \eta\, [u, c], t' = \mathsf{enter}_e(t)\}\ \textbf{in} \\
&\textbf{let}\ T' = \{r' \mid t \in \eta\, [u, c], r \in \eta\, [\mathsf{ret}_f, \mathsf{enter}_e(t)], \\
&\quad r' = \mathsf{combine}_{x,v}\, t\, r\}\ \textbf{in} \\
&(\overrightarrow{S}, T')
\end{aligned}$$

The set of traces reaching the start of the called function f is determined by extending each trace t reaching $[u, c]$ via $\mathsf{enter}_e(t)$. The transformation $\mathsf{enter}_e(t)$ extends the trace t by assigning the values of the arguments a_i to the corresponding formal parameters and initializes the local variables of f. In case any of the formal parameters is of type jump buffer, its value is set to err, as jump buffers may only be set via `setjmp`. Each resulting trace t' is then side effected to the start node of f st_f in context t'. The contribution to the program node v in the current context is then determined via $\mathsf{combine}_{x,v}\, t\, r$, where r is obtained from the return node ret_f of f in the context f was entered with. The function $\mathsf{combine}_{x,v}\, t\, r$ then extends the trace r with a configuration (v, σ''), where the new state σ'' is obtained from the last state σ in r by assigning the value of `retv` to x and then removing the local variables of f. Thus the function $\mathsf{combine}_{x,v}\, t\, r$ is defined by

$$\begin{aligned}
&\mathsf{combine}_{x,v}\, t\, r = \textbf{let}\ (u, \sigma) = \mathsf{last}(r)\ \textbf{in} \\
&\textbf{let}\ \sigma' = \sigma \oplus \{x \mapsto [\![\mathtt{retv}]\!]\, \sigma\}\ \textbf{in} \\
&\textbf{let}\ \sigma'' = \mathsf{remove\_locals}(\sigma', t)\ \textbf{in} \\
&r \boxplus (v, \sigma'')
\end{aligned}$$

Guard If the action a is a guard with the conditional expression g, then

$$\begin{aligned}
&[\![e, c]\!]\, \eta = \textbf{let}\ T = \eta\, [u, c]\ \textbf{in} \\
&\textbf{let}\ T' = \{t \boxplus (v, \sigma) \mid t \in T, \mathsf{last}(t) = (u, \sigma), [\![g]\!]\, \sigma \neq 0\}\ \textbf{in} \\
&(\emptyset, T')
\end{aligned}$$

For each trace t that reaches the program point u in context c and its last state σ, it is checked whether g evaluates to a nonzero value. In this case the trace is extended by one configuration with an unchanged state σ. The set T' of all such traces is the contribution to the next program point.

4 `longjmp` with stack-unwinding

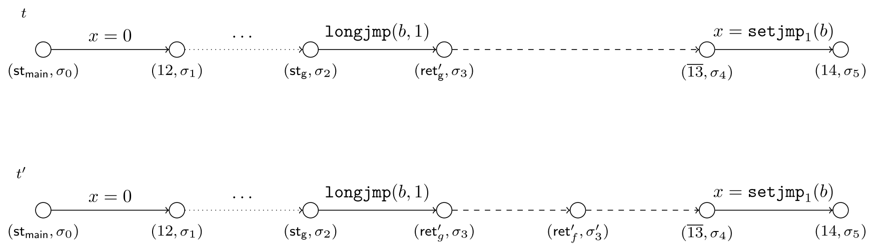
The trace-based semantics provided so far closely resembles the semantics of C with `setjmp` and `longjmp`. We will now present an alternative trace-based semantics where `longjmp`s are *localized* to jumps within function bodies only. To achieve this, *stack-unwinding* is introduced. The alternative semantics is again formalized as the least solution of a constraint system. This constraint system C' then will form the basis of our analysis. The constraints of C' agree with the constraints of C, except for the edges (u, a, v) where the action a is a `longjmp` or a function call. For that modification, the constraint system C is extended with further unknowns of the form $[\mathsf{ret}'_f, c]$, where ret'_f represents the artificial node of a function f signifying irregular returns, and $c \in \mathcal{T}$ is the context in which the function f has been reached. For these new unknowns, the mapping η now contains sets of pairs (bv, t), where bv is a jump buffer value, and t is a trace propagated from a `longjmp` to bv.

Example 2
Consider the example program given in Fig. 3 with `setjmp/longjmp`, together with the traces reaching the program point after the `setjmp` with an irregular return via `longjmp`. The trace reaching that program point in the least solution of C' contains a step for unwinding each function from the stack, whereas the unwinding is done in one step in the corresponding trace in the least solution of C.

For `longjmp`s and function calls, the new constraints for edges (u, a, v) in context c are defined in the following, whereas the right-hand sides of other constraints remain unchanged.

```
1   int g(){
2     longjmp(b, 1);
3     return 2;
4   }
5   int f() {
6     int z = 3;
7     z = g();
8     return z;
9   }
10  int main() {
11    volatile int x = 0;
12    int y = 0;
13    x = setjmp(b);
14    y = f();
15    return x;
16  }
```

(a) Source code


t
$x=0$
$\ldots$
$\texttt{longjmp}(b,1)$
$x = \texttt{setjmp}_1(b)$
$(\mathsf{st}_{\mathsf{main}}, \sigma_0)$
$(12, \sigma_1)$
$(\mathsf{st}_{\mathsf{g}}, \sigma_2)$
$(\mathsf{ret}'_g, \sigma_3)$
$(\overline{13}, \sigma_4)$
$(14, \sigma_5)$
t'
$x=0$
$\ldots$
$\texttt{longjmp}(b,1)$
$x = \texttt{setjmp}_1(b)$
$(\mathsf{st}_{\mathsf{main}}, \sigma_0)$
$(12, \sigma_1)$
$(\mathsf{st}_{\mathsf{g}}, \sigma_2)$
$(\mathsf{ret}'_g, \sigma_3)$
$(\mathsf{ret}'_f, \sigma'_3)$
$(\overline{13}, \sigma_4)$
$(14, \sigma_5)$

(b) Traces t and t' reaching program point 14 in the least solution of $\mathcal{C}$ and $\mathcal{C}'$, respectively.

Fig. 3 Example code using `setjmp/longjmp`, together with traces t and t' contained in the least solutions of C and C', respectively, reaching program point 14. There a node with number n indicates the program point at the start of line n. An edge label indicates the action that led to the sink configuration. The trace t directly returns from g to the target node, whereas the trace t' performs the unwinding of function calls f and g in separate steps.

`longjmp` If the action a is given by `longjmp(b,x)`, then

$$\begin{aligned}
&[\![e,c]\!]\,\eta = \textbf{let } V = \{t \mid t \in \eta\,[u,c], \mathsf{valid}(b,t)\} \textbf{ in}\\
&\quad \textbf{let } H = \{([\bar{u}',c], (r \boxplus (\bar{u}',\sigma'))) \mid\\
&\qquad r \in V, \mathsf{last}(r) = (u,\sigma),\\
&\qquad \sigma' = \sigma \oplus \{\texttt{retv} \mapsto [\![(x == 0)\ ?\ 1 : x]\!]\,\sigma\},\\
&\qquad \langle \bar{u}', t'\rangle = [\![b]\!]\,\sigma, \mathsf{here}_{u,c}\langle \bar{u}', t'\rangle\}\\
&\quad \textbf{in}\\
&\quad \textbf{let } O = \{([\mathsf{ret}'_{\mathsf{func}\,u}, c], (bv, r \boxplus (\mathsf{ret}'_{\mathsf{func}\,u}, \sigma'))) \mid\\
&\qquad r \in V, \mathsf{last}(r) = (u,\sigma),\\
&\qquad \sigma' = \sigma \oplus \{\texttt{retv} \mapsto [\![(x == 0)\ ?\ 1 : x]\!]\,\sigma\},\\
&\qquad bv = [\![b]\!]\,\sigma, \neg\mathsf{here}_{u,c}\,bv\}\\
&\quad \textbf{in}\\
&\quad (\overrightarrow{H \cup O}, \emptyset)
\end{aligned}$$

In case the jump buffer value to be jumped to is *local*, the definition has not changed, i.e., the relation H is defined as before.

A difference occurs for traces for which the jump buffer b has a value bv outside the current function invocation. In this case, it is prolonged by one step (instead of two before) by a configuration $(\mathsf{ret}'_{\mathsf{func}\,u}, \sigma')$, where the node $\mathsf{ret}'_{\mathsf{func}\,u}$ signifies the irregular return from the current function, and σ' is obtained by updating the value of the variable `retv`. The pair of bv and the extended trace then is side-effected to the unknown $[\mathsf{ret}'_{\mathsf{func}\,u}, c]$. As in the previous semantics, the empty set of traces is contributed to $[v,c]$, indicating that the successor program point cannot be reached via this edge.

Function calls If the action a has the form `x = f(a`$_1$`, ..., a`$_k$`)`, then

$$\begin{aligned}
&[\![e,c]\!]\,\eta = \textbf{let } S = \{([\mathsf{st}_f, t'], t') \mid\\
&\qquad t \in \eta\,[u,c], t' = \mathsf{enter}_e(t)\}\\
&\quad \textbf{in}\\
&\quad \textbf{let } T' = \{r' \mid t \in \eta\,[u,c], r \in \eta\,[\mathsf{ret}_f, \mathsf{enter}_e(t)],\\
&\qquad r' = \mathsf{combine}_{x,v}\,t\,r\}\\
&\quad \textbf{in}\\
&\quad \textbf{let } H = \{([\bar{u}',c], r') \mid t \in \eta\,[u,c],\\
&\qquad (\langle \bar{u}', t'\rangle, r) \in \eta\,[\mathsf{ret}'_f, \mathsf{enter}_e(t)],\\
&\qquad r' = \mathsf{combine}_{\texttt{retv},\bar{u}'}\,t\,r, \mathsf{here}_{u,c}\langle \bar{u}', t'\rangle\}\\
&\quad \textbf{in}\\
&\quad \textbf{let } O = \{([\mathsf{ret}'_{\mathsf{func}\,u}, c], (\langle \bar{u}', t'\rangle, r')) \mid t \in \eta\,[u,c],\\
&\qquad (\langle \bar{u}', t'\rangle, r) \in \eta\,[\mathsf{ret}'_f, \mathsf{enter}_e(t)],\\
&\qquad r' = \mathsf{combine}_{\texttt{retv},\mathsf{ret}'_{\mathsf{func}\,u}}\,t\,r, \neg\mathsf{here}_{u,c}\langle \bar{u}', t'\rangle\}\\
&\quad \textbf{in}\\
&\quad (\overrightarrow{S \cup H \cup O}, T')
\end{aligned}$$

The side-effecting of start states to the start points of the called functions and regular returns from the call are handled as before. Now, additionally, irregular returns of the called function must be handled. There are two kinds of irregular returns: those with a target inside the current function invocation and those with a target outside. The first kind of returns is collected in H, whereas O collects the second kind.

In H, for each trace t reaching u in context c, each tuple (bv, r) from the unknown $[\mathsf{ret}'_f, \mathsf{enter}_e(t)]$ is considered. Let $bv = \langle \bar{u}', t'\rangle$. Via the predicate $\mathsf{here}_{u,c}\langle \bar{u}', t'\rangle$, it is ensured that only the traces with a target in the current function invocation are considered. For such a pair (bv, r), the trace r is extended via $\mathsf{combine}_{\texttt{retv},\bar{u}'}$ by one step with a configuration

$(\bar{u}', \sigma')$, where the local state σ' results from the previous local state by assigning the value of `retv` to x; moreover, all locals that are not on the stack in the current function invocation are removed. The resulting trace is then is put into relation with the unknown $[\bar{u}', c]$.

In O, for each trace t reaching the program point u in the context c, those tuples $(\langle \bar{u}', t' \rangle, r)$ in $[\mathsf{ret}'_f, \mathsf{enter}_e(t)]$ are considered for which the target $\langle \bar{u}', t' \rangle$ is not in the current function invocation. Each such trace r is extended via the call $\mathsf{combine}_{\mathtt{retv}, \mathsf{ret}'_{\mathsf{func}\,u}}\, t\, r$, which appends a configuration $(\mathsf{ret}'_{\mathsf{func}\,u}, \sigma'')$ to r, where σ'' is obtained from the last state of r by removing the local variables of the functions that are not on the stack at the current function invocation in t. The resulting trace is then related to the unknown $[\mathsf{ret}'_{\mathsf{func}\,u}, c]$, where the returns via `longjmp` of the current function and context are collected. From the sets S, H and O, the side-effects for the edge are generated.

5 Equivalence of C and C'

Intuitively, the traces in the semantics formalized by C' can be obtained from the traces from C by elaborating the `longjmp`s from nested function calls into a sequence of irregular function returns followed by a function-local jump to some start node $\bar{u}$ of a $\mathtt{setjmp}_1$ edge. Conversely, traces of C can be obtained from traces of C' by collapsing these sequences into two steps, where the first updates the value of `retv`, and the second performs the interprocedural jump to the same start $\bar{u}$.

To formalize this bijection between traces, we define a function $i(t)$that, given a trace t from C, returns the corresponding trace in C' reflecting the same program execution. Recall that certain subtraces are used to construct jump buffer values. Therefore $i(t)$ is defined via mutual recursion with a function $j(\sigma)$ transforming program states:

$$\begin{array}{lcl} i(st_{\mathsf{main}}, \sigma) & = & (st_{\mathsf{main}}, j(\sigma)) \\ i(t \boxplus (v, \sigma)) & = & i(t) \boxplus (v, j(\sigma)) \quad \text{if } v \notin \mathcal{B} \\ i(t \boxplus (\mathsf{ret}'_{g_0}, \sigma_0) \boxplus (\bar{u}, \sigma)) & = & \\ \quad i(t) \boxplus (\mathsf{ret}'_{g_0}, j(\sigma_0)) \boxplus & & \\ \quad (\mathsf{ret}'_{g_1}, j(\sigma_1)) \boxplus \ldots \boxplus (\mathsf{ret}'_{g_r}, j(\sigma_r)) \boxplus & & \\ \quad (\bar{u}, j(\sigma)) & & \end{array}$$

where $g_0, \ldots, g_r$ is the sequence of functions to be returned from to arrive at the function invocation of the jump target $\bar{u}$, and for $k \geq 0$, σ_{k+1} is obtained from σ_k by removing the locals of g_k. The mapping $j(\sigma)$ is given by

$$j(\sigma) = \sigma \oplus \{x \mapsto \langle \bar{u}, i(t) \rangle \mid \sigma(x) = \langle \bar{u}, t \rangle, x \in \mathcal{V}\}$$

When lifting the mapping i between traces of C and C' to a mapping between least solutions of the two constraint systems, we must take into account that the set of unknowns may differ in two ways:

- C has unknowns $[u, c]$ for context traces c, whereas the corresponding unknowns of C' are $[u, i(c)]$; moreover,
- C' uses the set A of auxiliary unknowns $[\mathsf{ret}'_f, c]$, $f \in \mathsf{Funs}$, c a context trace of C'.

When establishing a correspondence, we concentrate on the values of unknowns $[u, c]$ with u different from ret'_f. Let η be an assignment of sets of traces to the unknowns of C. Then we define the mapping $I(\eta)$ for the nonauxiliary unknowns of C' by

$$I(\eta)[u, i(c)] = \{i(t) \mid t \in \eta\,[u, c]\}.$$

Proposition 1
Let η be the least solution of C, and let η' be the least solution of C'. Then

1. *for all unknowns $[u, c']$ of C' that are not in A,*
$$\eta'\,[u, c'] \sqsubseteq I(\eta)[u, c'];$$
2. *For any $(\langle \bar{u}', i(t') \rangle, r') \in \eta'[\mathsf{ret}'_g, i(c)]$, the following holds:*
   (a) *There is at least one occurrence of a node u in r' such that there is an edge of the form $(u, \mathtt{longjmp}(b, x), v) \in \mathcal{E}$. We denote the last such u in r' as u_0 and denote the prefix of r up to that last occurrence of u_0 as r'_0.*
   (b) *$r'_0 \in \eta'\,[u_0, \mathit{context}(r'_0)]$, and $\mathsf{valid}(b, r'_0)$, and for $(u_0, \sigma_0) = \mathit{last}(r'_0)$, it holds that $[\![b]\!]\,\sigma_0 = \langle \bar{u}', i(t') \rangle$ and $\mathit{func}\,u_0 \neq \mathit{func}\,\bar{u}'$.*
   (c) *Let for some $n \in \mathbb{N}^+$, $g_n, \ldots, g_0$ be the sequence of functions to be returned from in r'_0 to arrive at the function invocation at the jump target $\bar{u}'$. For some $n_0, 1 \leq n_0 \leq n$, it holds that $r' = s_{n_0}$, where s_m for $0 \leq m \leq n_0$ is defined as follows:*
$$\begin{array}{l} s_0 = r'_0 \boxplus (\mathit{ret}'_{\mathit{func}\,u_0}, \sigma_1) \\ s_{m+1} = s_m \boxplus (\mathit{ret}'_{g_{m+1}}, \sigma_{m+1}) \end{array}$$
   where $\sigma_1 = \sigma_0 \oplus \{\mathtt{retv} \mapsto [\![(x == 0\,?\,1 : x)]\!]\,\sigma_0\}$, and σ_{m+1} is obtained from σ by removing the locals of function g_m. It holds that $g_0 = \mathit{func}\,u_0$ and $g = g_{n_0}$.

Proof
We proceed by fixed-point induction on C'. Let η'_k be the kth iterate of the constraint system C' for $k \in \mathbb{N}$. For $k = 0$, $\eta'_k\,[u, c] = \emptyset$ for all u, c. Therefore claims 1 and 2 hold for η'_0. Now assume that the claims hold for η_k, and we show that the claims hold for η_{k+1}. Here we only elaborate on the side effects performed by function calls for irregular returns. Let e be an edge $e = (u, a, v)$, where the action is a function call $x = f(a_1, \ldots, a_k)$, and let $i(c)$ be some concrete context. For claim 2, we have to consider the side effects collected in the set O of the right-hand side. The right-hand side may

cause some jump buffer value and trace $(\langle \bar{u}', t' \rangle, r')$ to be side-effected to $[\mathsf{ret}'_{\mathsf{func}\,u}, i(c)]$, i.e.,

$$(\langle \bar{u}', t' \rangle, r') \in (\llbracket (u, a, v), i(c) \rrbracket \eta'_k)_1 \, [\mathsf{ret}'_{\mathsf{func}\,u}, i(c)].$$

Such a side effect is only performed under the condition that the following holds for some t:

$$t \in \eta'_k \, [u, i(c)] \wedge (\langle \bar{u}', t' \rangle, r) \in \eta'_k \, [\mathsf{ret}'_f, \mathsf{enter}_e(t)] \wedge \\ r' = \mathsf{combine}_{\mathtt{retv}, \mathsf{ret}'_{\mathsf{func}\,u}} \, t \, r \wedge \neg \mathsf{here}_{u, i(c)} \langle \bar{u}', t' \rangle.$$

According to the induction hypothesis (claim 2a), there is a prefix r'_0 of r such that the last node u_0 in r'_0 is the start of a `longjmp` edge. As r' prolongs the trace r by one step with $\mathsf{ret}'_{\mathsf{func}\,u}$ as the node, claim 2a also holds for r' with the same r'_0. By induction hypothesis (claim 2b) there is some c_0 such that $r'_0 \in \eta'_k \, [u_0, i(c_0)]$, and there is some $r_0 \in \eta \, [u_0, c_0]$ with $i(r_0) = r'_0$ and $\mathsf{valid}(b, r_0)$. As r'_0 is also contained in $\eta'_{k+1} \, [u_0, i(c_0)]$, claim 2b holds.

For r and r'_0, let n, n_0, and $g_n, \ldots, g_0$ be as described in 2c with $f = g_{n_0}$ and $r = s_{n_0}$, given by induction hypothesis. By construction, at the current edge e, the function g_{n_0} is called, and $\mathsf{func}\,u$ is on the stack in r. As $\neg \mathsf{here}_{u, i(c)} \langle \bar{u}', t' \rangle$ holds, the target of the jump is not in the current function invocation, and $\mathsf{func}\,u$ needs to be popped from the stack to perform the `longjmp` in r'_0. Therefore claim 2c holds for r' for $n_0 + 1$, and $g_{n_0+1} = \mathsf{func}\,u$.

For claim 1, we only consider the side effects for propagating irregular returns into the current function invocation. The right-hand side $\llbracket (u, a, v), i(c) \rrbracket$ may cause a trace r' to be side-effected to $[\bar{u}', i(c)]$, i.e.,

$$r' \in (\llbracket (u, a, v), i(c) \rrbracket \, \eta'_k)_1 \, [\bar{u}', i(c)].$$

Such a side-effect is only performed under the condition that there is some t such that

$$t \in \eta'_k \, [u, i(c)] \wedge (\langle \bar{u}', \hat{t}' \rangle, r'') \in \eta'_k \, [\mathsf{ret}'_f, \mathsf{enter}_e(t)] \wedge \\ r' = \mathsf{combine}_{\mathtt{retv}, \bar{u}'} \, t \, r'' \wedge \mathsf{here}_{u, i(c)} \langle \bar{u}', \hat{t}' \rangle.$$

From the induction hypothesis it follows that for r'', there are r'_0, n, n_0, and $g_n, \ldots, g_0$ as described in claims 1 and 2 with $r' = s_{n_0}$. As the target of the jump is local to the function containing u, there are no further functions to return from besides $g_0, \ldots, g_{n_0}$, and therefore $r'' = s_n$ and $n_0 = n$. According to the induction hypothesis (claims 2b and 1), there are r_0 and c_0 such that $i(r_0) = r'_0$, and $r_0 \in \eta \, [u_0, c_0]$ with $\mathsf{valid}(b, r_0)$. Let us now consider the constraint in C with the right-hand side $\llbracket e_0, c_0 \rrbracket \, \eta$. As η is a solution of C, and all side-effects of the constraint are thus accounted for, we have

$$r_0 \boxplus (\mathsf{ret}'_{\mathsf{func}\,u_0}, \sigma'_0) \boxplus (\bar{u}', \sigma''_0) \in \eta \, [\bar{u}', \mathsf{context}(\hat{t})],$$

where $\sigma'_0 = \sigma_0 \oplus \{\mathtt{retv} \mapsto \llbracket (x == 0) \; ? \; 1 : x \rrbracket \, \sigma_0\}$, with $\mathsf{last}(r_0) = (u_0, \sigma_0)$, $\sigma''_0 = \mathsf{remove\_locals}(\sigma'_0, \hat{t})$, and $\langle \bar{u}', \hat{t} \rangle = \llbracket b \rrbracket \, \sigma_0$. As $r'_0 = i(r_0)$, it follows that $\mathsf{last}(r'_0) = (u_0, j(\sigma_0))$. According to 2b, $\llbracket b \rrbracket \, (j(\sigma_0)) = \langle \bar{u}', \hat{t}' \rangle$. It follows that $i(\hat{t}) = \hat{t}'$ and $\mathsf{context}(\hat{t}) = c$.

The sequence of functions to be returned from to arrive at the active invocation of the function containing target $\bar{u}'$ is exactly $g_0, \ldots, g_n$. It thus follows that

$$i(r_0 \boxplus (\mathsf{ret}'_{g_0}, \sigma'_0) \boxplus (\bar{u}', \sigma''_0)) = \\ i(r_0) \boxplus (\mathsf{ret}'_{g_0}, j(\sigma'_0)) \boxplus (\mathsf{ret}'_{g_1}, j(\sigma'_1)) \boxplus \cdots \boxplus \\ (\mathsf{ret}'_{g_n}, j(\sigma'_k)) \boxplus (\bar{u}', j(\sigma''_0)) \in I(\eta) \, [\bar{u}', i(c)],$$

where σ_{k+1} is obtained from σ_k by removing the locals of the function g_k from σ_k. From $r' = s_n \boxplus (\bar{u}', j(\sigma''_0)) = i(r_0 \boxplus (\mathsf{ret}'_{\mathsf{func}\,u_0}, \sigma'_0) \boxplus (\bar{u}', \sigma''_0))$ it follows that

$$r' \in I(\eta) \, [\bar{u}', i(c)]. \qquad \square$$

Proposition 2
Let η' be the least solution of C'. Then the mapping η defined by

$$\eta \, [u, c] = \{t \mid i(t) \in \eta' \, [u, i(c)]\}$$

is a solution of C.

Proof
Let η' be the least solution of C', and let η be defined as in Proposition 2. It suffices to verify that the mapping η satisfies all constraints of C. Here we only elaborate on the constraints for `longjmp`s. For `longjmp`s, it must be verified that the adapted constraints introduced by C' simulate the single constraint for a `longjmp` in C by means of the auxiliaries. For an edge $e = (u, \mathtt{longjmp}(b, x), v)$ in some context c, consider the set of side effects yielded by the right-hand side $\overrightarrow{\llbracket e, c \rrbracket \, \eta}$ in C for the mapping η. This set is of the form $H \cup O$, where H and O collect all jump targets inside and outside to the current invocation, respectively.

Local jumps Let $([\bar{u}', c], (r \boxplus (\bar{u}', \sigma'))) \in H$. By the definition of H,

$$r \in \eta \, [u, c] \wedge \mathsf{valid}(b, r) \wedge \mathsf{last}(r) = (u, \sigma) \wedge \\ \sigma' = \sigma \oplus \{\mathtt{retv} \mapsto \llbracket (x == 0) \; ? \; 1 : x \rrbracket \sigma\} \wedge \\ \langle \bar{u}', t' \rangle = \llbracket b \rrbracket \, \sigma \wedge \mathsf{here}_{u,c} \langle \bar{u}', t' \rangle.$$

By the definition of η and the mapping i it follows that

$$i(r) \in \eta' \, [u, i(c)] \wedge \mathsf{last}(i(r)) = (u, j(\sigma)) \wedge \\ \langle \bar{u}', i(t') \rangle = \llbracket b \rrbracket \, (j(\sigma)) \wedge \mathsf{here}_{u, i(c)} \langle \bar{u}', i(t') \rangle.$$

Additionally, the property $\mathsf{valid}(b, i(r))$ is preserved from $\mathsf{valid}(b, r)$. We observe that

$$\begin{aligned} j(\sigma') &= j(\sigma \oplus \{\mathtt{retv} \mapsto \llbracket (x == 0) \; ? \; 1 : x \rrbracket \, \sigma\}) \\ &= j(\sigma) \oplus \{\mathtt{retv} \mapsto \llbracket (x == 0) \; ? \; 1 : x \rrbracket \, (j(\sigma))\}. \end{aligned}$$

Thus the value $(i(r) \boxplus (\bar{u'}, j(\sigma')))$ is contained in $\eta'[\bar{u'}, i(c)]$. Accordingly, $(r \boxplus (\bar{u'}, \sigma')) \in \eta[\bar{u'}, c]$, implying that every side effect in H is accounted for by η.

Nonlocal jumps Now consider the contributions that are collected in the set O of jumps outside the current function invocation. Assume that $([\bar{u}', c'], r') \in O$ with $r' = r \boxplus (\mathsf{ret}'_{\mathsf{func}\,u}, \sigma') \boxplus (\bar{u}', \sigma'')$. Then, according to the definition of O,

$$\begin{aligned} r \in \eta\,[u, c] \wedge \mathsf{valid}(b, r) \wedge \mathsf{last}(r) = (u, \sigma) \wedge \\ \sigma' = \sigma \oplus \{\mathtt{retv} \mapsto [\![(x == 0)\ ?\ 1 : x]\!]\,\sigma\} \wedge \\ \langle \bar{u}', t' \rangle = [\![b]\!]\,\sigma \ \wedge \sigma'' = \mathsf{remove\_locals}(\sigma', t') \wedge \\ c' = \mathsf{context}(t') \wedge \neg\mathsf{here}_{u,c}\langle \bar{u}', t' \rangle. \end{aligned} \tag{1}$$

We verify that the constraints in C' realizing the `longjmp` and the function calls occurring in $i(r)$ for which the callee is to be popped from the stack extend and propagate $i(r)$ appropriately to the target location. The right-hand side for the constraint for the `longjmp` is given by $[\![(u, a, v), i(c)]\!]$. We set $u_0 = u$ and $c_0 = i(c)$. The constraints for the function calls are identified by the function call nodes $u_k, u_{k-1}, \ldots, u_1$ occurring in the trace $i(r)$ after the prefix $i(t')$ for which the callee neither returned regularly nor via `longjmp`. For $m \in \{1, \ldots, k\}$, we refer to the trace that is the prefix of $i(r)$ up to the last occurrence of u_m as t_m. The right-hand sides of the constraints corresponding to the function call at u_m are identified via the edge $e_m = (u_m, a_m, v_m) \in \mathcal{E}$ and the context c_m of the trace t_m. We remark that $t_m \in \eta'\,[u_m, c_m]$ and $c_k = \mathsf{context}(i(t')) = i(\mathsf{context}(t')) = i(c')$.

Now, for $m \in \{0, \ldots, k-1\}$, we prove that

$$(\langle \bar{u}', i(t') \rangle, r_m) \in \eta'\,[\mathsf{ret}'_{\mathsf{func}\,u_m}, c_m],$$

where $r_0 = i(r) \boxplus (\mathsf{ret}'_{\mathsf{func}\,u_0}, \sigma'_0)$, $r_{m+1} = r_m \boxplus (\mathsf{ret}'_{\mathsf{func}\,u_{m+1}}, \sigma'_{m+1})$, with $\sigma'_0 = j(\sigma')$, and $\sigma'_{m+1} = \mathsf{combine}_{\mathtt{retv}, \mathsf{ret}'_{\mathsf{func}\,u_{m+1}}}\ t_{m+1}\, r_m$ with $m \in \{0, \ldots, k-2\}$.

Base case Consider $m = 0$. From Eq. (1) it follows by the definitions of η and i and from $\mathsf{valid}(b, r)$ that

$$\begin{aligned} & i(r) \in \eta'\,[u, i(c)] \wedge \mathsf{valid}(b, i(r)) \ \wedge \mathsf{last}(i(r)) = (u, j(\sigma)) \\ & \wedge \langle \bar{u}', i(t') \rangle = [\![b]\!]\, j(\sigma) \ \wedge \neg\mathsf{here}_{u, i(c)}\langle \bar{u}', i(t') \rangle. \end{aligned}$$

Therefore, due to the side effect made by the right-hand side $[\![(u, \mathtt{longjmp}(b, x), v), i(c)]\!]\,\eta'$, we have that

$$(\langle \bar{u}', i(t') \rangle, i(r) \boxplus (\mathsf{ret}'_{\mathsf{func}\,u}, j(\sigma'))) \in \eta'\,[\mathsf{ret}'_{\mathsf{func}\,u}, i(c)].$$

Induction step Assume that the statement holds for $m \in \{0, \ldots, k-2\}$. By construction, $c_m = \mathsf{enter}_{e_{m+1}}(t_{m+1})$. Together with the induction hypothesis, we thus have that $(\langle \bar{u}', i(t') \rangle, r_m) \in \eta'\,[\mathsf{ret}'_{\mathsf{func}\,u_m}, \mathsf{enter}_{e_{m+1}}(t_{m+1})]$. Further, it holds that $\neg\mathsf{here}_{u_{m+1}, c_{m+1}}\langle \bar{u}', t' \rangle$, as $\bar{u}'$ is not contained in the function $\mathsf{func}\ u_{m+1}$. Thus the constraint with the right-hand side $[\![e_{m+1}, c_{m+1}]\!]\,\eta'$ requires that

$$(\langle \bar{u}', t' \rangle, r_{m+1}) \in \eta'\,[\mathsf{ret}'_{\mathsf{func}\,u_{m+1}}, c_{m+1}]$$

with $r' = \mathsf{combine}_{\mathtt{retv}, \mathsf{ret}'_{\mathsf{func}\,u}}\ t_{m+1}\, r_m$. This concludes the proof by induction.

It remains to show that the constraint for the function call at u_k in context c_k extends the trace appropriately and propagates it to the unknown $[\bar{u}', c_k]$. From the statement proven by induction and with $\mathsf{enter}_{e_k}\, t_k = c_{k-1}$ it follows that

$$(\langle \bar{u}', i(t') \rangle, r_{k-1}) \in \eta'\,[\mathsf{ret}'_{u_{k-1}}, \mathsf{enter}_{e_k}\, t_k].$$

We observe that by construction of u_k and c_k the predicate $\mathsf{here}_{u_k, c_k}\langle \bar{u}', i(t') \rangle$ holds. The constraint with right-hand side $[\![e_k, c_k]\!]\,\eta'$ then requires that $r_k \in \eta'\,[\bar{u}', c_k]$, where $r_k = \mathsf{combine}_{\mathtt{retv}, \bar{u}'}\, t_k\, r_{k-1}$. From $i(c') = c_k$ and $i(r') = r_k$ and by construction of η it follows that $i(r') \in \eta'\,[\bar{u}', i(c')]$, and therefore $r' \in \eta\,[\bar{u}', c']$. □

Theorem 1
Let η and η' be the least solutions of C and C', respectively. Then

$$\eta'\,[u, c'] = I(\eta)[u, c']$$

for all unknowns $[u, c']$ of C' that are not in A.

Proof
Follows from Propositions 1 and 2. We calculate for $c' = i(c)$:

$$\begin{array}{rcl} \eta'\,[u, i(c)] & \sqsubseteq & I(\eta)[u, i(c)] \\ & = & \{i(t) \mid t \in \eta\,[u, c]\} \\ & \sqsubseteq & \{i(t) \mid t \in \{t \mid i(t) \in \eta'\,[u, i(c)]\}\} \\ & = & \eta'\,[u, i(c)], \end{array}$$

from which the equality follows. □

6 Indeterminate local variables

The preceding definitions of the concrete semantics have ignored the possibility of encountering poisoned variables, i.e., nonvolatile variables that have been modified between a `setjmp` and a corresponding call to `longjmp`. To restrict the semantics such that traces containing accesses to poisonous variables are not prolonged, we adapt the semantics of transfer functions $[\![e, c]\!]$ in our constraint systems to a semantics $[\![e, c]\!]'$ as follows:

$$\begin{aligned} & [\![e, c]\!]'\,\eta = \mathbf{let}\ T = \eta\,[u, c]\ \mathbf{in} \\ & \quad \mathbf{let}\ T' = \{t \mid t \in T, \neg\exists v \in \mathcal{V}, \\ & \quad\quad v \in \mathsf{read\_vars}(e, t) \wedge v \in \mathsf{poisonous}(t)\}\ \mathbf{in} \\ & \quad [\![e, c]\!]\,(\eta \oplus [u, c] \mapsto T') \end{aligned}$$

where $\mathsf{read\_vars}(e,t)$ yields the set of program variables that are read at edge e for trace t, and the set $\mathsf{poisonous(t)}$ yields the set of nonvolatile local variables that were written in t between a `longjmp` and the corresponding `setjmp` where the buffer was set and have not been overwritten since. The function $\mathsf{poisonous}$ may be defined inductively on traces. The set T' is the set of traces reaching $[u,c]$ such that *no read* of a poisoned variable occurs in the step. Altogether, $[\![e,c]\!]'$ then behaves like the original $[\![e,c]\!]$, but the latter now receives the set T' as the new value for the unknown $[u,c]$.

7 Base analysis

In the following, we build on the least solution of the constraint system C' as our reference semantics for our analysis. These therefore are also formalized using side-effecting constraint systems over a set of unknowns $\mathcal{X}$ taking values. Instead of sets of traces, however, the unknowns now take *abstract* values from some complete lattice $\mathbb{D}$. We assume that there is a monotonic mapping γ from $\mathbb{D}$ to sets of traces with the understanding that $\gamma\, d$ provides the set of all well-formed traces *described* by d. In particular, $\gamma\,\top$ should be the set of all traces, and $\gamma\,\bot$ should be the empty set.

For context-sensitive analysis, the unknowns are pairs of program points and *abstract* contexts β, which we denote by $[u,\beta]$. Let $\mathbb{C}$ be some set of contexts, each of them again describing a set of concrete contexts, i.e., concrete traces. We require a concretization function $\gamma_{\mathbb{C}}$ for contexts analogous to γ for abstract values. The analysis is relative to some initial context $\bullet \in \mathbb{C}$, in which `main` is analyzed, and some abstract value d_0, so that $\gamma_{\mathbb{C}}\,\bullet \supseteq \mathsf{init}$ and $\gamma\, d_0 \supseteq \mathsf{init}$. The constraints of the system have the form

$$\begin{array}{rcll} \eta\,[st_{main},\bullet] & \sqsupseteq & d_0; & \\ (\eta,\eta\,[v,\beta]) & \sqsupseteq & [\![e,\beta]\!]^\sharp\,\eta, & e=(u,a,v)\in\mathcal{E}. \end{array}$$

For simplicity, we assume that $\mathbb{D}$ is a nonrelational mapping from program variables to abstract values of some complete lattice. This base analysis provides us with constraints for assignments, guards, and function calls. Here we briefly recall how right-hand sides for function calls of the base analysis are constructed. For a control-flow edge $e=(u,x=f(a_1,\ldots,a_n),v)$ and context $\beta\in\mathbb{C}$, the right-hand side $[\![e,\beta]\!]^\sharp\,\eta$ is given by

$$\begin{array}{l} [\![e,\beta]\!]^\sharp\,\eta = \textbf{let}\;\sigma\; = \eta\,[u,\beta]\;\textbf{in} \\ \qquad \textbf{let}\;\langle\beta',\sigma'\rangle = \mathsf{enter}_e^\sharp\,\sigma\;\textbf{in} \\ \qquad \textbf{let}\;\sigma'' = \mathsf{combine}_x^\sharp\,\sigma\,(\eta\,[\mathsf{ret}_f,\beta])\;\textbf{in} \\ \qquad (\{[\mathsf{st}_f,\beta']\mapsto\sigma'\},\sigma'') \end{array}$$

Here $\mathsf{enter}_e^\sharp$ takes the local state of the caller and yields a pair. The first component β' is the calling context for f; its second component is an abstract entry state for the function f obtained by, e.g., removing unreachable locals of the caller and assigning (abstract) values for the actuals to the formals. The entry state is then side-effected to the unknown $[\mathsf{st}_f,\beta]$ corresponding to the entry node of f in context β.

The function $\mathsf{combine}_x^\sharp$ takes as its first argument the local state of the caller and as its second argument the state computed for the endpoint of function f in context β, i.e., the value of unknown $[\mathsf{ret}_f,\beta]$, and combines these states into the abstract state after the function call. This entails, e.g., removing local variables of f and assigning the value of `retv` to the variable x on the left-hand side of the call. We assume that the following holds for traces t, t' and abstract states $\sigma,\sigma'\in\mathbb{D}$ with $t\in\gamma(\sigma)$, $t'\in\gamma(\sigma')$:

$$\begin{array}{rcl} \mathsf{enter}_e\,t & \in & \gamma(\mathsf{enter}_e^\sharp\,\sigma)_2, \\ \mathsf{enter}_e\,t & \in & \gamma_{\mathbb{C}}(\mathsf{enter}_e^\sharp\,\sigma)_1, \\ \mathsf{combine}_{x,v}\,t\,t' & \in & \gamma(\mathsf{combine}_x^\sharp\,\sigma\,\sigma')\text{ for all } v\in\mathcal{N}, \end{array}$$

where a subscript n denotes taking the nth value of the tuple. For the analysis of programs with `setjmp/longjmp`, we now enhance the base analysis by extending the abstract domain $\mathbb{D}$ to express auxiliary information and lifting all right-hand sides accordingly. New transfer functions corresponding to `setjmp` and `longjmp` are generically assembled from the abstract effects of assignments, $\mathsf{enter}_e^\sharp$, and $\mathsf{combine}_{x,v}^\sharp$ as provided by the base analysis.

8 Analysis of `Setjmp/Longjmp`

For analyzing `setjmp/longjmp`, we identify three tasks:

P1 Tracking, which targets *may* legally be jumped to, i.e., keeping track of all potential invocations of `setjmp` where the containing call has not returned yet.
P2 Tracking the values of variables of type `jmp_buf`.
P3 Using information from **P1** and **P2**, propagating states from `longjmp` to `setjmp`.

To realize items **P1** and **P2**, we first introduce an abstract domain $\mathbb{D}_{\mathsf{buf}}$ to track values of variables of type `jmp_buf`. The domain consists of sets of pairs of barred nodes and contexts as well as an error value $\mathsf{err}^\sharp$, i.e., $\mathbb{D}_{\mathsf{buf}} = 2^{\mathcal{B}\times\mathbb{C}\cup\{\mathsf{err}^\sharp\}}$ ordered by $\subseteq$. For $B\in\mathbb{D}_{\mathsf{buf}}$, the set $\gamma\,B$ of described jump buffer values consists of err if $\mathsf{err}^\sharp\in B$, together with all $\langle\bar{u},t\rangle$ where there is some abstract jump buffer target $\langle\bar{u},\beta\rangle\in B$ such that the context of t is described by the abstract context β, i.e., $\mathsf{context}\,t\in\gamma_{\mathbb{C}}\,\beta$.

For **P1**, we track the set of possibly valid jump targets in an auxiliary local variable legal, which receives values from $\mathbb{D}_{\mathsf{buf}}$. Let $\bar{\gamma}$ and $\bar{\gamma}_{\mathbb{C}}$ denote the concretization functions used by the extended analysis for abstract values and contexts, respectively. Recall that now abstract program states $\sigma^\sharp$ are mappings from program variables as well as the auxiliary variable legal to abstract values. Then the concretization $\bar{\gamma}\,\sigma^\sharp$ of an abstract state $\sigma^\sharp$ is given by the set of all traces t such that (1) $t \in \gamma\,(\sigma^\sharp)$, where one ignores the auxiliary variable legal, and (2) all values of valid jump buffers are included in $\gamma\,(\sigma^\sharp\,\mathsf{legal})$. Similarly, $\bar{\gamma}_{\mathbb{C}}$ is obtained from $\gamma_{\mathbb{C}}$.

A call to `setjmp` adds the pair $\langle \bar{u}, \beta\rangle$ for current node u and calling context β to the value of legal. The set of legal jump targets is passed to callees, but upon combine, the set from the caller is restored, as the callee has already returned: any jump targets inside the callee are no longer legal. The abstract functions $\mathsf{enter}^\sharp_e$ and $\mathsf{combine}^\sharp_x$ are extended accordingly.

For realizing **P2**, the base analysis is assumed to track for each variable of type `jmp_buf` a value from $\mathbb{D}_{\mathsf{buf}}$ along the lines of an off-the-shelf *values-of-variables* analysis. Care must be taken that the error value $\mathsf{err}^\sharp$ must be added when a jump buffer is written by some operation different from $\texttt{setjmp}_0$. This is taken care of by adapting $\mathsf{enter}^\sharp_e$ and the abstract constraints for assignments. For this analysis of values of jump buffer variables b, we demand that each abstract jump buffer target $\langle \bar{u}', \beta'\rangle$ is preserved if the buffer b is not certainly overwritten in the concrete. We further demand that for all jump buffer variables b, the functions $\mathsf{enter}^\sharp_e$ and $\mathsf{combine}^\sharp_x$ maintain their jump buffer values for any call edge e and `int` variable x: The transfer function for $e = (u, \texttt{x=setjmp}_0\texttt{(b)}, v)$ in context β adds the value $\{\langle \bar{u}, \beta\rangle\}$ to the values of `b` and legal and sets the variable `x` to `0`:

$$\begin{aligned}
&[\![e,\beta]\!]^\sharp\ \eta = \textbf{let}\ \sigma = \eta\ [u,\beta]\ \textbf{in}\\
&\ \textbf{let}\ B = [\![\&b]\!]^\sharp\ \sigma\ \textbf{in}\\
&\ \textbf{let}\ \sigma' = \sigma \oplus \{x \mapsto [\![0]\!]^\sharp, \mathsf{legal} \mapsto \sigma\,\mathsf{legal} \cup \langle \bar{u},\beta\rangle\}\ \textbf{in}\\
&\ \left(\emptyset, \textstyle\bigsqcup_{\&j\in B}(\sigma' \oplus \{j \mapsto \{\langle \bar{u},\beta\rangle\}\})\right)
\end{aligned}$$

Here $[\![\&b]\!]^\sharp\ \sigma$ provides us with a set B of addresses of jump buffers returned by the abstract evaluation of the jump buffer expression b.

The right-hand side for $\texttt{setjmp}_1$ in calling context β takes its argument from some unknown $[\bar{u},\beta]$ where all states from arriving `longjmp`s have been collected via side effects. Recall that the return value of `longjmp` is tracked via the return variable `retv`. The constraint for $\texttt{setjmp}_1$ then is the same as that for the assignment `x=retv`.

Now we can give the right-hand sides of the enhanced analysis to accomplish **P3**. For function calls, the single return node ret_f is complemented with a secondary node ret'_f as introduced by the concrete semantics for function f to be exited via a `longjmp` in f itself or in transitively called functions. The abstract values at unknowns for ret'_f consist of sets of pairs of current jump target and abstract state.

`Longjmps` Assume that $e = (u, \texttt{longjmp(b,x)}, v)$. Then

$$\begin{aligned}
&[\![e,\beta]\!]^\sharp\ \eta = \textbf{let}\ \sigma = \eta\ [u,\beta]\ \textbf{in}\\
&\ \textbf{let}\ T = [\![b]\!]^\sharp\ \sigma\ \textbf{in}\\
&\ \textbf{let}\ L = T \cap (\sigma\,\mathsf{legal})\ \textbf{in}\\
&\ \textbf{let}\ \sigma' = [\![\texttt{retv = (x==0 ? 1 : x)}]\!]^\sharp\ \sigma\ \textbf{in}\\
&\ \textbf{let}\ \rho_{\mathrm{h}} = \{[\bar{u}',\beta] \mapsto \sigma' \mid \langle \bar{u}',\beta'\rangle \in L \wedge \mathsf{here}^\sharp_{u,\beta}\langle \bar{u}',\beta'\rangle\}\ \textbf{in}\\
&\ \textbf{let}\ \rho_{\mathrm{o}} = \{[\mathsf{ret}'_{\mathsf{func}\,u},\beta] \mapsto \{(bv,\sigma') \mid bv \in L\}\}\ \textbf{in}\\
&\ (\rho_{\mathrm{h}} \cup \rho_{\mathrm{o}}, \bot)
\end{aligned}$$

where the predicate $\mathsf{here}^\sharp_{u,\beta}$ is an abstract version of the predicate $\mathsf{here}_{u,c}$ in the concrete semantics. Accordingly, it is defined by

$$\mathsf{here}^\sharp_{u,\beta}\langle \bar{u}',\beta'\rangle := (\mathsf{func}\,u = \mathsf{func}\,\bar{u}') \wedge (\beta = \beta').$$

The predicate checks if its argument is a jump target that is local to the current function and calling context.

Thus the right-hand side for a `longjmp` first evaluates the expression b to a set of jump targets. A warning may be emitted if it cannot be excluded that the jump buffer has the error value. To be able to warn about jumps to invalid jump targets, i.e., where the enclosing function may already have returned, our overapproximation of legal jump targets is not sufficient. This may be remedied by introducing path- and context-sensitivity in the values of legal, as discussed later. Then the return state σ' is prepared. The dedicated variable retv for the return values of function calls also receives the values returned by the call to `setjmp`. Care is taken to set the value of `retv` to `1` should `x` be `0`. If the analysis cannot exclude `0` for `x`, an appropriate warning is issued (omitted for clarity). For those jump targets $\langle \bar{u}',\beta'\rangle$ *within* the given function and abstract calling context, a side effect to the unknown $[\bar{u}',\beta']$ for arriving via `longjmp` is produced. For all jump targets including those *outside* the current function and abstract calling context, a side effect to the unknown $[\mathsf{ret}'_{\mathsf{func}\,u},\beta]$ for returning from the current function via `longjmp` is produced. We remark that one could restrict this mapping to abstract jump targets only for which the concretization does not intersect with the concretization of jump targets in the current function and context. The contribution to the control-flow successor of the `longjmp` statement is $\bot$, as the call to `longjmp` does not return.

Function calls For the right-hand sides corresponding to function calls, on top of accounting for a normal return in the manner introduced in Sect. 7, potentially occurring `longjmp`s must be handled. These may occur in the function itself or within some nested function call. Again, a case distinction is required on whether the jump target may occur within the current function and context. Let

$e = (u, x = f(a_1, \ldots, a_n), v)$ with $\mathsf{func}\, u = g$. Then

$$\begin{aligned}
&[\![e, \beta]\!]^\sharp\ \eta = \mathbf{let}\ \sigma\ = \eta\ [u, \beta]\ \mathbf{in}\\
&\mathbf{let}\ \langle \beta, \sigma' \rangle = \mathsf{enter}^\sharp_e\ \sigma\ \mathbf{in}\\
&\mathbf{let}\ \sigma'' = \mathsf{combine}^\sharp_x\ \sigma\ (\eta\ [\mathsf{ret}_f, \beta])\ \mathbf{in}\\
&\mathbf{let}\ J = \{(bv, \mathsf{combine}^\sharp_{\mathsf{retv}}\ \sigma\ \sigma') \mid (bv, \sigma') \in \eta\ [\mathsf{ret}'_f, \beta]\}\ \mathbf{in}\\
&\mathbf{let}\ \rho_\mathrm{h} = \{[\bar{u}', \beta] \mapsto \sigma_{\bar{u}',\beta} \mid \mathsf{here}^\sharp_{u,\beta} \langle \bar{u}', \beta' \rangle\ \wedge\\
&\qquad \sigma_{\bar{u}',\beta} = \bigsqcup \{\sigma' \mid (\langle \bar{u}', \beta' \rangle, \sigma') \in J\}\ \mathbf{in}\\
&\mathbf{let}\ \rho_\mathrm{o} = \{[\mathsf{ret}'_g, \beta] \mapsto J\}\ \mathbf{in}\\
&(\{[\mathsf{st}_f, \beta] \mapsto \sigma'\} \cup \rho_\mathrm{h} \cup \rho_\mathrm{o}, \sigma'')
\end{aligned}$$

The right-hand side now accesses the unknown $[\mathsf{ret}'_f, \beta]$ accounting for leaving f via `longjmp`. The value of this unknown is a set of pairs of jump targets and local states. Recall that in these local states σ', the value supplied as the second argument to the call of `longjmp` has been written to the variable `retv`. This value is preserved by the call to $\mathsf{combine}^\sharp_{\texttt{retv}}$. The resulting set of pairs of jump targets and local states of the current function g is collected in J. Those local states for jump targets $\langle \bar{u}', \beta \rangle$ in J that are inside the current function and context are side-effected to the unknowns $[\bar{u}', \beta]$. The pairs of targets and values from J are propagated to the dedicated unknown $[\mathsf{ret}'_g, \beta]$ for `longjmp`ing out of g.

The analysis thus performs a *stack-unwinding* as introduced in the concrete semantics given by the constraint system C' to account for all effects of function calls further up the stack prior to a call to `longjmp`. Notably, this includes modifications to local variables that have become visible to other functions and may have been modified by them.

Example 3
Assuming that the context for the call to `bar` in `main` in Fig. 1 is $\beta_{\texttt{bar}}$, the set of legal jump targets is set to $L = \{\langle \bar{u}_{14}, \beta_{\texttt{bar}} \rangle\}$ by the `setjmp` in line 14. L is passed to the call to `foo` in context $\beta_{\texttt{foo}}$. At the call of `longjmp` in line 8, the abstract value for `errorhandler` is $\{\langle \bar{u}_{14}, \beta_{\texttt{bar}} \rangle\}$. As it contains only elements in L, all jump targets are legal. The analysis then assigns the abstract value for `err` to `retv`, resulting in some abstract state σ'. As the jump target is not inside the current function, the value $\{p\}$ with $p = (\langle \bar{u}_{14}, \beta_{\texttt{bar}} \rangle, \sigma')$ is side-effected to the special return unknown $[\mathsf{ret}'_{\texttt{foo}}, \beta_{\texttt{foo}}]$, and any code after the `longjmp` is marked as dead (although, here, there is none). Now consider the call to `foo` in line 21 of function `bar`. For leaving `foo` via `longjmp`, the value of $[\mathsf{ret}'_{\texttt{foo}}, \beta_{\texttt{foo}}]$ is accessed. As the jump target in p is in the current function and context, the state σ'' obtained by combining σ' with the local state at line 21 is side-effected to unknown $[\bar{u}_{14}, \beta_{\texttt{bar}}]$. This state, modified by the effect of the assignment `x = retv`, accounts for returning to line 14 via `longjmp` from `foo`. For the call to `longjmp` in line 30, on the other hand, the jump target is illegal, and an appropriate warning is produced.

We formalize the soundness of the proposed extended analysis with respect to the trace semantics C' given in Sect. 3. We refer to the constraint system of the analysis as $C^\sharp$. Since the unknowns between the concrete and abstract constraint system differ, we establish a relationship between assignments of unknowns of the concrete and abstract constraint system by means of the following definition.

Definition 1
Let η' be a mapping for the concrete constraint system C', and let $\eta^\sharp$ be a mapping for the abstract constraint system. We say that η' is *described* by $\eta^\sharp$ iff for every function f, concrete context c, and abstract context β with $c \in \bar{\gamma}_\mathbb{C}\, \beta \cap \bar{\gamma}\, (\eta^\sharp\ [\mathsf{st}_f, \beta])$, it follows that

$$\eta'\ [v, c] \subseteq \bar{\gamma}\, (\eta^\sharp\ [v, \beta])$$

for all program points v of f, including the extra nodes in $\mathbb{B}$ and ret'_f.

Theorem 2
Let η be the least solution of C', and let $\eta^\sharp$ be a solution of $C^\sharp$. Then η is described by $\eta^\sharp$.

The proof of this theorem is by fixpoint induction on the constraint system C', where the key ingredient is that the set init of concrete initial states is contained in $\bar{\gamma}_\mathbb{C} \bullet \cap\, \bar{\gamma}\, d_0$, i.e., is described both by the abstract calling context $\bullet$ for the initial call to main and the initial abstract state d_0 for $[\mathsf{st}_{\mathsf{main}}, \bullet]$, and that this relationship is preserved by each right-hand side of C' and $C^\sharp$.

Proposition 3
Let $e = (u, a, v)$ be a control-flow edge of the program, and let c be a concrete context with $c \in \bar{\gamma}_\mathbb{C}\, \beta$. Assume that $(\rho, T) = [\![e, c]\!]\, \eta'$ and $(\rho^\sharp, \sigma^\sharp) = [\![e, \beta]\!]^\sharp\ \eta^\sharp$, where η' is described by $\bar{\gamma}\, \eta^\sharp$. Then $T \subseteq \bar{\gamma}\, \sigma^\sharp$. Moreover, for ρ, $\rho^\sharp$, we have:

1. *If a is the call $x = f(a_1, \ldots, a_k)$, then*
   (a) *$\rho\ [\bar{u}', c] \subseteq \bar{\gamma}\, (\rho^\sharp\ [\bar{u}', \beta])$ for all $\bar{u}' \in \mathbb{B}$ occurring in the current function $\mathsf{func}\, u$;*
   (b) *$\rho\ [\mathsf{ret}'_{\mathsf{func}\, u}, c] \subseteq \bar{\gamma}\, (\rho^\sharp\ [\mathsf{ret}'_{\mathsf{func}\, u}, \beta])$;*
   (c) *$\rho\ [\mathsf{st}_f, \mathsf{enter}_e\, t] \subseteq \bar{\gamma}\, (\rho^\sharp\ [\mathsf{st}_f, (\mathsf{enter}^\sharp_e (\eta^\sharp\ [u, \beta]))_1])$, where $t \in \eta'\ [u, c]$.*
2. *If a equals `longjmp`(b, x), then*
   (a) *$\rho\ [\bar{u}', c] \subseteq \bar{\gamma}\, (\rho^\sharp\ [\bar{u}', \beta])$ for all $\bar{u}' \in \mathbb{B}$ occurring in the current function $\mathsf{func}\, u$;*
   (b) *$\rho\ [\mathsf{ret}'_{\mathsf{func}\, u}, c] \subseteq \bar{\gamma}\, (\rho^\sharp\ [\mathsf{ret}'_{\mathsf{func}\, u}, \beta])$.*

Our analysis handles correct usages of `setjmp/longjmp`, detects dead code following a call to `longjmp` **(H)**, and warns if the second argument to `longjmp` may be $\mathbb{0}$ **(G)**. It detects invocations of `longjmp` for buffers that have not been correctly set by `setjmp` by means of tracking the abstract error value $\mathsf{err}^\sharp$ **(A)**. To warn about jumps into functions that have already returned, the abstract value of the auxiliary variable legal in our implementation is tracked *context-* and *path-sensitively* **(B)** (a similar technique has, e.g., been applied to keep sets of held mutexes apart [26]). To detect when `longjmp` is called from a different thread than `setjmp` **(C)**, an analysis of thread *id*s is required as, e.g., provided in [24].

To achieve **(E)**, the *values-of-variables* analysis tracks whether the value of a variable of type `jmp_buf` was set via an invocation of `setjmp` or not. To achieve **(F)**, first, the concrete semantics must be extended to deal with variably sized objects on the stack. Then it suffices to collect for each `setjmp` the set of scopes defining variably sized objects and then track for each `longjmp` and each call, possibly terminated by means of a `longjmp`, the set of scopes that have potentially been left. Our current implementation does not perform this detailed analysis, but instead warns when `setjmp`s happen in scopes defining variably sized objects.

9 Detecting indeterminate local variables

It remains to detect whether nonvolatile local variables have been modified since the call to `setjmp` **(D)**. We take a three-pronged approach by combing three different taint analyses (T_{intra}, T_{inter}, and T_{poison}). Inside the function that performs the call to `setjmp`, T_{intra} tracks the set of potentially modified nonvolatile locals of automatic storage duration jump target-sensitively. For a called function f, T_{inter} tracks the set of those passed to and potentially modified by f, independently of jump targets. This is justified since few local variables are usually passed to called functions. T_{inter} employs an interprocedural taint-analysis, which we do not detail here. Lastly, T_{poison} tracks variables that have Indeterminate Value after a `longjmp` (these variables are said to be *poisonous*) and warns upon access to such variables.

Let $\mathcal{V}_{\mathtt{f}}$ the set of *nonvolatile* local variables of automatic storage duration occurring in a function `f`. For the *intraprocedural* jump target-sensitive analysis T_{intra}, we employ an extra domain $\mathbb{D}_{\text{taint}} = \mathcal{B} \to 2^{\mathcal{V}_{\mathtt{f}}}$, consisting of mappings from barred nodes to sets of local variables that potentially have been written since the corresponding jump target was set. The ordering on $\mathbb{D}_{\text{taint}}$ is given by $\subseteq$ lifted to partial maps, i.e., $\mu_1 \subseteq \mu_2$ iff $\mu_1\, a \subseteq \mu_2\, a$ whenever $\mu_1\, a$ is defined. Functions are entered with the empty partial mapping. When a `setjmp` is encountered at $[u, \alpha]$, the binding $\bar{u} \mapsto \emptyset$ is added to the mapping. Whenever a relevant local is modified, it is added to *each* set in the mapping. For each function call, T_{inter} yields a set of relevant locals possibly modified during the call, which is added to each set in the mapping.

T_{poison} maintains for every program point and context, a set of possibly poisonous variables P. Consider a `longjmp` at program point u in context α, and assume that T_{intra} has determined for $[u, \alpha]$ the mapping μ. For every barred node $\bar{u}_j$ within the current function where μ is defined, P additionally receives the set $\mu\, \bar{u}_j$. Function calls are treated similarly. Starting from the jump targets, the set P of possibly poisonous variables is propagated. Whenever a variable in P is *definitely* assigned to, it is removed from P, as the indeterminate value is overwritten. If a variable in P *may* be accessed, then the analyzer warns that an indeterminate value may be read.

Example 4
After the `setjmp` in line 14 of `bar` in Fig. 1, the map from barred nodes to potentially written local variables is set to $\{\bar{u}_{14} \mapsto \emptyset\}$. In line 20, the nonvolatile local variable `logpath` is modified, resulting in $\{\bar{u}_{14} \mapsto \{$`logpath`$\}\}$. After `foo` is left via `longjmp`, the set of poisonous variables is $\{$`logpath`$\}$. Since `logpath` is not overwritten before being accessed on line 17, a warning is produced.

We have thus enhanced the analysis with taint analysis to not only remain sound in the presence of `setjmp/longjmp`, but also to flag all possible issues (**A**)–(**H**), thus enabling developers to use `setjmp/longjmp` without fear of falling short.

10 Implementation and evaluation

We have implemented the analysis within the analysis framework Goblint[2] for multithreaded C. Our implementation supports C without the restrictions made for the formalization. In particular, recursion, dynamic memory allocation. and dynamic function calls are supported. We leveraged existing domains and the support for *path-sensitivity* (in the return values of calls to `setjmp`). No changes to solvers were necessary. The implementation itself performs all analyses jointly. We evaluated it in two ways: First, we extracted challenging usages of `setjmp/longjmp` from real-world programs and crafted 45 litmus tests for semantic issues of `setjmp/longjmp` and verified that the analyzer passes these. These may serve as sanity checks for future analyzers.

From the set of litmus tests, 37 programs, together with expected verdicts, were submitted as tasks for the software verification competition SV-COMP 2024 [3]. Some litmus tests could not be used as SV-COMP tasks, as they contained types of faults not supported by the SV-COMP rules. On the submitted tasks, Goblint achieved the highest score among the participating tools at SV-COMP 2024. Nine participating tools yielded unsound verdicts on at least some of these tasks, and nine further tools could not solve any of them.

[2] https://goblint.in.tum.de and https://github.com/goblint/analyzer.

The libpng[3] is a C library heavily using setjmp/longjmp. Several bugs in uses of this library have been reported.[4] To check real-world applicability, we analyzed the pngtest program,[5] which comes with that library. The analysis takes 76 minutes on a machine with an Intel Xeon 8260 CPU, where it uses one processor core. The number of warnings for setjmp/longjmp related issues is reasonable: Two warnings about accesses to a single distinct poisonous variables are reported, as well as one warning about jumping to an unknown jump buffer, one warning about jumping to a jump buffer with a value not set by setjmp, and two warnings about jumping to potentially invalid targets. Manual inspection reveals most of the warnings to be spurious: Goblint's loss of precision here is due to libpng's heavy usage of dynamically allocated memory objects with structs containing jump buffers, pointers to jump buffers, etc. To get rid of the spurious warnings, a more sophisticated heap analysis would be required. The warnings about accesses to the poisonous variable row_buf, however, are indicative of Undefined Behavior. We could not observe any misbehavior of the binary compiled with Gcc, because the address of this pointer variable escapes to an external function preventing assignment to a register. It is, however, dangerous to rely on such incidental properties to ensure program safety. The clobbering does manifest when making a natural change to the program using single-row read and write functions.[6] In fact, this usage pattern has resulted in a memory leak in an older version of ImageMagick [18]. That bug, when injected into pngtest, is correctly detected by Goblint. For a less heap-intensive benchmark, we considered the exceptions library proposed by Roberts [19]. The use of the library can result in Undefined Behavior if nonvolatile variables are updated between the *try*-clause and the *catch*-clause as these macros are translated into setjmp/longjmp.[7] Goblint detects cases where the client code elicits Undefined Behavior, whereas Gcc 12 and Clang 14 (with -O2) clobber such variables – without issuing warnings. We also evaluated the performance penalty of enabling this analysis on the SV-COMP 2022 benchmarks and observed a slowdown of 17%. This can be remedied though by a preanalysis that only enables this analysis for programs where longjmp is statically called or referenced via pointer. Our implementation, the litmus tests, and the larger programs are publicly available.[8]

[3] http://www.libpng.org/pub/png/libpng.html.

[4] e.g., in ImageMagick commits 75fc6 and e88c1. Also, [18].

[5] Around 7000 logical lines of live code.

[6] https://github.com/glennrp/libpng/issues/496.

[7] https://github.com/cs50/spl/issues/24.

[8] https://github.com/goblint/bench/tree/longjmp/setjmp.

11 Related work

Feng et al. [9] give a Hoare-style framework for the verification of assembly code including support for stack-based control abstractions such as setjmp/longjmp. We aim for *automatic* techniques. Nonlocal control flow has also been considered for an analysis of Python code [10] (elaborated on by Monat [16]) and in particular to handle generators. The analysis does not rely on side-effecting constraint systems. Instead, it iterates over the syntax and account for nonlocal flow, which, unlike in our setting, also includes break and continue in loops by partitioning according to *flow tokens*. States associated with some of these tokens are incorporated into the current flow when the respective program point is encountered. Like in our setting, targets in which execution is to be resumed need to be tracked. The challenges, though, are different as control-flow for generators is more structured: Upon a call to next, execution resumes at the beginning of the generator or its last call to yield. Upon calling yield in the generator, control is returned to the caller of next, and execution continues at that program point. It is not immediately obvious how to craft iteration over the syntax for dealing with setjmp/longjmp: A call to longjmp may transfer control to program points that are not static control-flow successors of any of the calls containing longjmps.

The analysis of exceptions in languages such as Java or C++ is also closely related; see, e.g., Chang and Choi [5] for a recent comprehensive survey. As our approach intertwines the analysis of nonlocal control flow with other analyses, e.g., of points-to information, it is most closely related to the class of *Combined Exceptional Control Flow Analyses* identified in the survey. However, analyzing setjmp/longjmp poses challenges not faced when analyzing C++ or Java programs where nonlocal control flow is well structured. Wilson [28] provides an account of handling setjmp/longjmp in a context-sensitive summary-based pointer analysis: His approach introduces a second function summary accounting for leaving the function via longjmp. Unlike our approach, he does not track values of jump buffers and assumes that a longjmp may jump to any setjmp further up the callstack. The analysis only endeavors to compute sound pointer information in the presence of correct usages of setjmp and does not detect possible misuses of the feature, as ours does. Other work [11] claims to use a similar technique to Wilson but does not elaborate on details.

12 Conclusions and future work

We have provided a novel technique for lifting static analysis to support setjmp/longjmp. Dynamic control-flow is dealt with by side effects in transfer functions. We formalized the behavior of setjmp/longjmp in a subset of C in concrete semantics, where interprocedural `longjmps` are either performed directly or via stack-unwinding. The equivalence relationship between these two concrete semantics was given. Based on the second formulation of the concrete semantics, we discussed how to analyze programs containing correct usages of `setjmp/longjmp` and how to detect potentially incorrect usages. We have enhanced the static analyzer Goblint with the technique and evaluated it on challenging litmus tests and on real-world programs. In the future, tackling further "dark corners" of C that have also not seen widespread support from static analyzers, such as implementations of coroutines in C, may provide new and interesting challenges for crafting analyzers capturing more real-world aspects of programming.

Acknowledgements We thank Benjamin Bott for implementing an earlier prototype. This work was supported in part by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY, the Estonian Centre of Excellence in IT, funded by the European Regional Development Fund, and the Shota Rustaveli National Science Foundation of Georgia under the project FR-21-7973.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Apinis, K., Seidl, H., Vojdani, V.: side-effecting constraint systems: a Swiss army knife for program analysis. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Proceedings, Kyoto, Japan, December 11-13, 2012, Lecture Notes in Computer Science, vol. 7705, pp. 157–172. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-35182-2_12
2. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. Found. Trends Program. Lang. **2**(2–3), 71–190 (2015). https://doi.org/10.1561/2500000002
3. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems – 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Proceedings, Part III, Luxembourg City, Luxembourg, April 6–11, 2024. Lecture Notes in Computer Science, vol. 14572, pp. 299–329. Springer, Berlin (2024). https://doi.org/10.1007/978-3-031-57256-2_15
4. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation – 18th International Conference, VMCAI 2017, Proceedings, Paris, France, January 15–17, 2017. Lecture Notes in Computer Science, vol. 10145, pp. 112–130. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-52234-0_7
5. Chang, B., Choi, K.: A review on exception analysis. Inf. Softw. Technol. **77**, 1–16 (2016). https://doi.org/10.1016/j.infsof.2016.05.003
6. Christakis, M., Bird, C.: What developers want and need from program analysis: an empirical study. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016, pp. 332–343. ACM, New York (2016). https://doi.org/10.1145/2970276.2970347
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, pp. 238–252. ACM, New York (1977). https://doi.org/10.1145/512950.512973
8. Erhard, J., Schinabeck, J.F., Schwarz, M., Seidl, H.: When to stop going down the rabbit hole: taming context-sensitivity on the fly. In: Monat, R., Rubio-González, C. (eds.) Proceedings of the 13th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, SOAP 2024, Copenhagen, Denmark. ACM, New York (2024). To appear
9. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: Schwartzbach, M.I., Ball, T. (eds.) Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11–14, 2006, pp. 401–414. ACM, New York (2006). https://doi.org/10.1145/1133981.1134028
10. Fromherz, A., Ouadjaout, A., Miné, A.: Static value analysis of python programs by abstract interpretation. In: Dutle, A., Muñoz, C.A., Narkawicz, A. (eds.) NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17–19, 2018. Proceedings, Lecture Notes in Computer Science, vol. 10811, pp. 185–202. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-77935-5_14
11. Hind, M., Pioli, A.: Evaluating the effectiveness of pointer alias analyses. Sci. Comput. Program. **39**(1), 31–55 (2001). https://doi.org/10.1016/S0167-6423(00)00014-9
12. Leroy, X.: The CompCert C verified compiler – documentation and user's manual – version 3.12 (2022). Tech. rep
13. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015). https://doi.org/10.1145/2644805
14. MITRE: CVE-2018-14876. (2018). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14876. [accessed 09-March-2023]
15. MITRE: CVE-2013-1441. (2013). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1441. [Accessed 09-March-2023]
16. Monat, R.: Static type and value analysis by abstract interpretation of Python programs with native C libraries. (analyse statique, de type et de valeur, par interprétation abstraite, de programmes Python utilisant des librairies C). Ph.D. thesis, Sorbonne University, Paris, France (2021) https://tel.archives-ouvertes.fr/tel-03533030
17. Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa-c: improved verification for C programs, simple validation of correctness witnesses (competition contribution). In:

Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Proceedings, Part III, Luxembourg City, Luxembourg, April 6–11, 2024. Lecture Notes in Computer Science, vol. 14572, pp. 387–392. Springer, Berlin (2024). https://doi.org/10.1007/978-3-031-57256-2_26

18. Patrakov, A.: Dangers of setjmp()/longjmp() (2009). https://patrakov.blogspot.com/2009/07/dangers-of-setjmplongjmp.html. Online; accessed 09-March-2023
19. Roberts, E.S.: Implementing exceptions in C. Tech. Rep. 40, Digital Equipment Corporation, Systems Research Center (1989)
20. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: abstract interpretation for memory safety and termination (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems – 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Proceedings, Part III, Luxembourg City, Luxembourg, April 6-11, 2024. Lecture Notes in Computer Science, vol. 14572, pp. 381–386. Springer, Berlin (2024). https://doi.org/10.1007/978-3-031-57256-2_25
21. Schubert, P.D., Hermann, B., Bodden, E.: Phasar: an interprocedural static analysis framework for C/C++. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems – 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings, Part II, Prague, Czech Republic, April 6–11, 2019. Lecture Notes in Computer Science, vol. 11428, pp. 393–410. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17465-1_22
22. Schwarz, M., Saan, S., Seidl, H., Apinis, K., Erhard, J., Vojdani, V.: Improving thread-modular abstract interpretation. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis – 28th International Symposium, SAS 2021, Proceedings, Chicago, IL, USA, October 17–19, 2021. Lecture Notes in Computer Science, vol. 12913, pp. 359–383. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-88806-0_18
23. Schwarz, M., Erhard, J., Vojdani, V., Saan, S., Seidl, H.: When long jumps fall short: control-flow tracking and misuse detection for non-local jumps in C. In: Ferrara, P., Hadarean, L. (eds.) Proceedings of the 12th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023, pp. 20–26. ACM, New York (2023). https://doi.org/10.1145/3589250.3596140
24. Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. In: Wies, T. (ed.) Programming Languages and Systems – 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Proceedings, Paris, France, April 22–27, 2023. Lecture Notes in Computer Science, vol. 13990, pp. 28–58. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30044-8_2
25. Seidl, H., Vene, V., Müller-Olm, M.: Global invariants for analysing multi-threaded applications. In: Proceedings – Estonian Academy of Sciences Physics Mathematics, vol. 52, pp. 413–436. Estonian Academy Publishers (2003)
26. Vojdani, V., Vene, V.: Goblint: path-sensitive data race analysis. Ann. Univ. Sci. Budapest., Sect. Comput. **30**, 141–155 (2009)
27. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the Goblint approach. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp. 391–402. ACM, New York (2016)
28. Wilson, R.P.: Efficient, context-sensitive pointer analysis for C programs. Ph.D. thesis, Stanford University (1997)