Sys: a Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code

Fraser Brown
Stanford University

Deian Stefan
UC San Diego

Dawson Engler
Stanford University

Abstract

We describe and evaluate an extensible bug-finding tool, Sys, designed to automatically find security bugs in huge codebases, even when easy-to-find bugs have been already picked clean by years of aggressive automatic checking. Sys uses a two-step approach to find such tricky errors. First, it breaks down large—tens of millions of lines—systems into small pieces using user-extensible static checkers to quickly find and mark *potential* errorsites. Second, it uses user-extensible symbolic execution to deeply examine these potential errorsites for actual bugs. Both the checkers and the system itself are small (6KLOC total). Sys is flexible, because users must be able to exploit domain- or system-specific knowledge in order to detect errors and suppress false positives in real codebases. Sys finds many security bugs (51 bugs, 43 confirmed) in well-checked code—the Chrome and Firefox web browsers—and code that some symbolic tools struggle with—the FreeBSD operating system. Sys's most interesting results include: an exploitable, cash bountied CVE in Chrome that was fixed in seven hours (and whose patch was backported in two days); a trio of bountied bugs with a CVE in Firefox; and a bountied bug in Chrome's audio support.

1 Introduction

This paper focuses on automatically finding security bugs, even in code where almost everything easy-to-find has been removed by continuous checking with every tool implementers could get their hands on. We check three systems in this category (§5): Google's Chrome browser, Mozilla's Firefox browser, and the SQLite database. Chrome fuzzers run 24/7 on over 25,000 machines [21] and are combined with dynamic tools that look for low-level memory errors, while Firefox runs at least six fuzzers just for its JavaScript engine [88]. Both browsers run modern static bug finding tools and both pay cash bounties for security vulnerabilities [51, 103]. Most extremely, the SQLite database, included in both Chrome and Firefox and thus checked with all of their methods, also includes three independent test suites with *100% branch coverage* which are run on many different architectures and configurations (32- and 64-bit, little and big endian, etc.) [22, 116].

Our new bug-finding system, Sys, was born out of our failure to find security bugs in Chrome and Firefox. One of our previous static tools—which looks for simple buggy patterns in source code, along the lines of [39, 60, 85, 121]—found only three security bugs in browsers [40]. As far as we could tell, most of the security bugs it was capable of finding were long gone. Our group's symbolic execution tool, KLEE [45]—which conceptually executes programs over all possible inputs, a powerful but expensive technique—simply couldn't scale to huge browser codebases off-the-shelf, and adapting such a complex tool was daunting. To address the drawbacks of both approaches, we combine them: static analysis, cheap and imprecise, achieves high recall in identifying possible errorsites, and symbolic analysis, expensive and thorough, achieves high precision in reasoning about those errorsites.

Sys first uses a static analysis pass to identify *potential* errorsites. This pass is not precise, and typically errs on the side of false positives over false negatives; Sys uses symbolic execution (symex) to "clean up" these results, as we discuss below. Users can write their own static extensions to identify potentially buggy snippets of code, or they can use Sys's existing passes to point symex in the right direction.

Next, Sys uses symbolic execution to reason deeply about each potential errorsite that static analysis (static) identifies. Symbolic execution generally provides high precision [47, 124]. For example, it can determine that a certain value must equal seven on one path and one hundred on the other. Fine-grained value reasoning means that symex can find bugs that static can't, but also makes symex routinely intractable, even for small programs: it reasons about *all possible values*, whereas simple static analysis reasons primarily about dataflows.

Sys sidesteps the symex bottleneck by only symbolically executing small snippets of code that the static analysis pass flags as potentially buggy. Intuitively, this works because many bugs only require a small amount of context to understand (e.g., finding an infinite loop may just require looking at the loop's body and header). This strategy is an adaption of underconstrained (UC) symbolic execution [63, 115], which improves the scalability of symex by executing individual functions instead of whole programs. Sys takes this a step further by only executing the snippets of code that static analysis identifies. Users can write their own symbolic analyses, or they can use Sys's existing analyses out-of-the-box.

Category	**Number**	**Reference**
Sec-high	1 (13 total bugs)	[1]
Sec-medium/moderate	4	[2–5]
Sec-low	4	[6–9]
Sec-other	3	[10, 11]
Bounty	3 (17 total bugs)	[1–3]
CVE	4 (18 total bugs)	[1, 3, 4, 7]
Security audits	2	[1, 12]
Patched functions	27	[1–3, 7, 11, 13, 14]
Patched bugs	16	[1–3, 7, 11, 13, 14]
Already patched	3	-
Mystery patch	5	-
Reported bugs	51	-
Confirmed bugs	43	-
False positives	18	-

Figure 1: This table summarizes the bugs Sys found. We do not double-count bugs or false positives that appear in both browsers. Browser vendors classify security bugs as [53, 105]: *sec-high*, e.g., bugs attackers can use to corrupt the browser's memory and hijack its control flow to, for instance, steal bank account information; *sec-medium*, e.g., bugs attackers can use to leak browser memory like login cookies; *sec-low*, bugs whose scope is limited, but would otherwise be considered higher severity. The bounty row indicates bugs that received cash rewards from the browsers in which they appeared, and the CVE row lists bugs that have been listed in a global vulnerability database. The security audits row lists bug reports that have prompted developers to "audit" their code for more instances of the bug we reported. Finally, the mystery patch row indicates patches that are unaccounted for: they patch bugs that Sys found, but because of backports, we can't tell *when* they were patched.

Finally, we designed Sys to be flexible, because real-world checking is a game of iterative hypothesis testing: in our experience, it takes many tries to express a property (e.g., use of uninitialized memory) correctly, and many more to suppress false positives—and both tasks often take advantage of ad hoc, program-specific information. We wanted Sys to combine the flexibility of a standard static checking framework (e.g., the Clang Static Analyzer [87, 151]) with the power of a symbolic execution engine.

The challenge of building a flexible symbolic checking tool is that symex is inherently complicated—it has to reason about each individual bit in the program under test—but flexibility requires that using and changing the system be easy. To address this challenge, we created an embedded domain-specific language (DSL) to abstract some of the complications of symbolic reasoning (§3). Users write symbolic checkers in the DSL. The entire Sys symbolic execution engine is written in the *same* DSL, which mechanically guarantees that users have the power to write arbitrary checkers, extend the system, or replace any part of it.

To the best of our knowledge, Sys is the first system to do symex on large, complex, automatically tested systems like browsers. The main contributions of this work are:

1. **An implementation of the system and five checkers that find good security bugs.** We used Sys to build five checkers for uninitialized memory, out-of-bounds access, and use-after-free (UAF) bugs. Sys found 51 bugs (Figure 1) in the Chrome browser, the Firefox browser, and the FreeBSD operating system, many in complicated C++ code. Sys discovered a group of 13 high-severity, exploitable SQLite bugs in the Chrome browser (CVE-2019-5827), which the SQLite author patched within seven hours; the patch was backported to Chrome within two days [1]. Sys also discovered a trio of bugs with a CVE in Firefox (CVE-2019-9805) [3], two more browser CVEs [4, 7], a user-after-free bug in Firefox [14], and a bountied bug in Chrome's audio support [2]. Finally, Sys is different enough from other checking tools that it can be used to check the checkers themselves (and vice versa): one of our bug reports [12] helped Firefox developers fix a configuration problem in the Coverity commercial checking tool. Sys is available at `https://sys.programming.systems`.

2. **An approach for scaling symbolic reasoning to huge codebases.** Fundamentally, full symbolic execution cannot scale to the browser. Sys's combination of static analysis and symbolic execution allows it to check entire browsers and get meaningful results in human time. The slowest checker covers all of Chrome in six hours on one (large) machine, and finds many real bugs.

3. **The design of a simple, extensible, DSL-based symbolic checking system that makes it possible to experiment with new checking techniques.** As a rough measure of complexity, Sys is only 6,042 lines of code (§3). It is easy to write new checkers (our static extensions are $<$ 280 LOC; our symbolic checkers are $\leq$ 110 LOC), add false positive suppression heuristics (§5.1,5.2), and even extend the core system (§3). As one example, building the static checking pass took a weekend. As others, we were able to add shadow memory to the system in a few hours and fewer than 20 lines of code, and Section 6 describes how someone with no checker-writing experience created a UAF checker that found a Firefox bug.

2 System overview

This section provides an overview of static checking and symbolic execution, and shows how Sys works in practice by walking through the steps it took to find a high-severity Chrome bug in their version of the SQLite database. Figure 2 shows the bug, an exploitable out-of-bounds write caused by integer overflow of an allocation size. To find the bug, users provide checkers (described below) and an LLVM IR file to check (e.g., generated by Clang from C source), and Sys outputs bug reports.

2.1 Finding the bug is hard

This bug requires precise reasoning about values (the overflow) and memory (the allocation), which is not the strong suit of most static tools. Other general bug finding methods aren't well positioned to find this bug, either.[1] Hitting the bug with

[1] A *specialized* tool like KINT [142], which only looks for integer overflows given code and annotations, is well positioned to find the bug.

```
     /* third_party/sqlite/patched/ext/fts3/fts3_write.c */
3398 const int nStat = p->nColumn+2;
     /* static extension stores allocation size of <a> */
3401 a = sqlite3_malloc( (sizeof(u32)+10)*nStat );
3402 if( a==0 ){
3403   *pRC = SQLITE_NOMEM;
3404   return;
3405 }
...
3414 if( sqlite3_step(pStmt)==SQLITE_ROW ) { ...
3415 } else{
     /* symbolic checker flags this <memset> as an error,
        the size passed in can be larger than <a> */
3419   memset(a, 0, sizeof(u32)*(nStat) );
3420 }
```

Figure 2: High-severity bug Sys found in SQLite: `nColumn` is a user-defined number of FTS3 columns, and attackers can craft a database with enough columns to overflow the allocation on line 3401 to a small value. Then, the big `memset` on line 3419 will be out-of-bounds [1].

a test case or an automatic dynamic tool is daunting, since SQLite is a large,[2] complex codebase even before being included in Chrome—and the path (the sequence of instructions) leading to the bug is complex, too. To reach it, you would have to start Chrome's WebSQL and make a database of the correct kind—among other things, you would need to create a virtual table instead of a regular table or view [1, 134]—which would require correctly exercising huge paths. *Even then*, the tool or test would have to stumble on the correct number of columns to trigger the bug. Randomly orchestrating these events is next to impossible. On the other hand, pure symbolic tools, which work in theory, are unable to handle massive codebases. Our group's previous tool, KLEE [45], does whole program symbolic execution on 10-30KLOC of C, not millions of lines of C++. UC-KLEE, our group's adaption to KLEE that scales by symbolically executing functions in isolation, would still need to be modified to check Chrome. Examining each of the $\approx$15 million Chrome functions would take about five CPU-years even if execution time were bounded to 10 seconds per function[3] (§6).

2.2 How Sys finds the bug

Sys makes it easy for users to identify potential bugs, and then lets them use symbolic reasoning (and their own application-specific knowledge) to check them. We walk through Sys's three steps below: (1) statically scanning the source and marking potential errors, (2) jumping to each marked location to check it symbolically, and (3) reasoning about state that Sys misses because it skips code.

Static Clients write small static extensions—similar to checkers that identify patterns in source code—to quickly scan all checked code and mark potential errorsites (Figure 4). Sys runs static extensions similarly to prior tools: it constructs

[2]Version 3.28.0 is 153,572 LOC according to cloc-1.8.

[3]UC-KLEE typically operates with a bound 30–60$\times$ longer than that, five to ten minutes.

```
 1 check :: Named Instruction -> Checker ()
 2 check instr = case instr of
 3
 4   -- Save the size of the object
 5   name := Call fName args | isAllocation fName -> do
 6     let allocSize = args !! 0
 7     saveSize name allocSize
 8
 9   -- Keep track of dependencies between LHS and RHS
10   -- variables of arithmatic instructions.
11   name := _ | isArith instr -> do
12     operands <- getOperands instr
13     forM_ operands $ addDep name
14
15   -- If an array index has some dependency on
16   -- an object's allocated size, report the path
17   name := GetElementPtr addr (arrInd:_) -> do
18     let addrName = nameOf addr
19     addrSize <- findSize addrName
20     when (isDep addrSize arrInd) $
21       reportPath arrSize arrInd
22
23   -- Otherwise do nothing
24   _ -> return ()
```

Figure 3: Simplified static extension for heap out-of-bounds errors. This checker looks for index operations (e.g., indexing (pictured) or `memset` (not pictured)) that are related to an object's allocated size.

a control flow graph from LLVM IR and then does a simple flow-sensitive traversal over it with the user's extension. Extensions are written in Haskell, and use a library of built-in routines to inspect and analyze the control flow graph. If a checker for a given bug already exists, clients can use that checker off the shelf.

Sys is subtly different from traditional static checkers, however. Traditional systems check program rules like "no deadlocks" by examining source code for buggy patterns like "two lock calls in a row with no unlock," and often aim to have a relatively low false positive rate. In contrast, Sys extensions should achieve high recall at identifying *possible errorsites*—which means that extensions are often crude, leaving serious reasoning (high precision) to the symbolic checker.

Figure 3 shows the static extension pass that marks the SQLite bug as a potential error. This extension looks for memory operations like `malloc(x)` and index operations like `memset(y)` where there is some relationship between `x` and `y`. Intuitively, the reason we look for this construct is that the dependency gives us enough information to compensate for unknown state (e.g., we probably won't know the values of `x` and `y`, but knowing their relationship can be enough to find bugs). The vast majority of these cases are not buggy, of course, but we'll use a symbolic checker to determine which are and which aren't later.

The extension itself uses Haskell's matching syntax (`case`) to do different actions depending on the IR instruction it is applied to. The conditional in lines 5-7 matches allocation calls and stores an association between the object's name and its allocated size. Then, the conditional on line 11 matches on any arithmetic instruction. It keeps track of dependencies

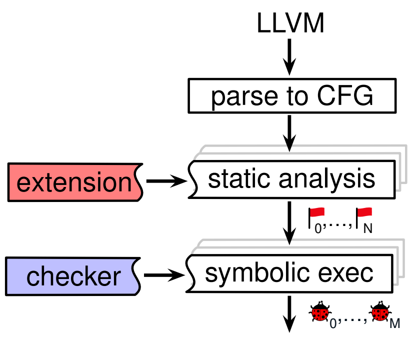

Figure 4: Developers provide the LLVM files they wish to check and a checker description in our framework. Their static extensions mark relevant program points, and their symbolic checkers jump to these points to symbolically execute in order to find bugs.

between variables in these instructions (e.g., `y = x + 1` would produce a dependency between `x` and `y`). Finally, when it matches on indexing operations (`GetElementPtr` on line 17), it marks any path where the index size has a dependency on the object's allocated size.

Symbolic The static pass produces potentially buggy paths, which Sys then feeds to the symbolic pass. This pass aims to achieve high precision at determining whether or not the bug actually exists by symbolically reasoning about all possible values on a given path. It: (1) automatically symbolically executes the entire path[4] and (2) applies the user's symbolic checker to the path.

Our tool, like other bit-accurate symbolic tools before it [47], aims to accurately track memory down to the level of a single bit—i.e., to assert *for sure* that a bit must be 0, must be 1, or may feasibly be either. Sys explores each potentially buggy code path individually, and it can explore a path either to termination or up to a window size. Each explored path has its own private copy of all memory locations it has read or written. As it advances down a path, Sys translates LLVM IR instructions into *constraints*, logical formulas that describe restrictions on values in the path. It also applies a user-supplied symbolic checker as it advances along the path (described below). Finally, Sys queries an SMT solver [37] to figure out if the path is possible. It receives either `UNSAT` if the path's constraints can never evaluate to true, or `SAT` if the path's constraints can.

The symbolic checker, in Figure 5, uses information that the static extension marked to figure out if an out-of-bounds write is possible. Specifically, its inputs on line 2 are the object size variable (`arrSize`) and index variable (`arrInd`) from the static extension. The symbolic checker is built from functions in Sys's symbolic DSL—`getName`, `toByes`, `isUge`—which

[4]The static phase gives the symbolic phase a complete path to execute, which can be a snippet of a loop or N unrolled iterations of a loop. Sys transforms the final path to single static assignment form to ensure that variables in loops are handled correctly.

```
1  symexCheck :: Name -> Name -> Symex ()
2  symexCheck arrSize arrInd = do
3
4      -- Turn the size into a symbolic bitvector
5      arrSizeSym <- getName arrSize
6      -- Turn the index into a symbolic bitvector
7      let indTy = typeOf arrInd
8      arrIndSym <- getName arrInd
9      arrIndSize <- toBytes indTy arrInd
10
11     -- Report a bug if the index size can be
12     -- larger than the allocation size
13     assert $ isUge byte arrIndSize arrSizeSym
```

Figure 5: A slightly simplified version of the heap out-of-bounds checker, without the symbolic false positive suppression.

are designed to easily and safely encode constraints about LLVM variables (§3). First, the checker translates its input variables into their symbolic representations (line 5), and uses `toBytes` to change the raw index value into its offset size in bytes (line 9). Then, it `asserts` that `arrIndSize` should be larger than `arrSizeSym`—indicating an out-of-bounds access (line 13). Mechanically, these SysDSL functions add new constraints to the logical formula, alongside the constraints Sys automatically adds when symbolically executing the path. Sys applies this particular checker once it has finished symbolically executing a path.

Symbolic checkers have control over which code to skip, where to start executing along the marked possible-error path, and even which functions to enter or avoid.[5] For example, the checker in Figure 5 runs on each function with a marked `malloc` call, and it runs after Sys has finished symbolically executing the whole path; other checkers match on specific LLVM IR instructions and run at different points along the path. Users write short configurations to tell Sys where and when to run their checkers.

The checker in Figure 5 looks at paths from the start of functions with marked `malloc` calls, but it could start either closer to `main` or closer to the `malloc`. The farther away it starts, the more values it knows, but the higher the cost of exploration. At one extreme, jumping right to the `malloc` call is cheap, but will lack all context before the call. At the other, starting at `main` and attempting to reach each site is the equivalent of traditional symbolic execution.

Unknown state Sys's approach of jumping over code to the error site is both its strength and its weakness. By skipping code, it also skips this code's constraints and side-effects, including memory allocation and initializations. Thus, the struggle is how to (1) make up fake copies of skipped state, and (2) ensure that missing constraints do not lead to explosions of false positives.

Sys makes up state using lazy allocation, similar to the UC-KLEE system [115]. If checked code dereferences a symbolic

[5]The checkers in this paper don't enter non-inlined function calls, but the implementation supports both behaviors.

location, Sys allocates memory for it and continues. This approach allows Sys to allocate exactly the locations that a path needs without any user interaction. However, allowing the contents of fake objects to be anything can cause false errors because of impossible paths and values. Sys doesn't drown us in false positives for four main reasons:

1. Sys's constraint solver eliminates all paths with internal contradictions (e.g., a path that requires a pointer to be both null and non-null); the only false positives that are left are due to the external environment (e.g., callers).

2. We use Sys to target specific errors instead of full functional correctness. As a result, many fake values that could potentially cause problems do not, since they don't affect the checked property in any way. For example, Sys will find the bug in Figure 2 even if the elided code does many different memory operations, as long as these operations don't touch the `nColumn` field.

3. Sys checkers can also account for undefined state in useful ways. For example, the `malloc` checker looks for out-of-bounds access in code snippets where there's a dependency between an object's allocation size and its index size. The dependency gives us important information—the relationship between an object's size and the index value—that allows us to find bugs without knowing what the object's size and index value actually *are*.

4. Large groups of false positives usually share a root cause, and Sys checkers can address that cause with ad hoc, checkers-specific tricks. For example, the checker that found the SQLite bug makes different assumptions about integer inputs compared to object fields: it assumes that integer inputs can be anything, while object fields have been vetted, and so must not be massive (§5.2). This one change eliminated many false positives.

Next, we discuss design decisions (§3–§4) and results (§5).

3 SysDSL design

Our goal was to build a symbolic checking system that was not just accurate, but also flexible enough to express checkers that could find bugs in huge codebases. Everything from prototyping checkers to hacking on the core system to suppressing false positives with ad hoc information—like the massive-value suppression in the previous section—had to be easy and fast. To that end, we aimed for a system that was:

1. **Domain specific**: at the highest level, the system should make *bug finding* easy. There should be direct, high-level ways to express both symbolic checks (e.g., "is `x` uninitialized") and ad hoc information (e.g., "all size arguments to `malloc` are greater than zero."). On the one hand, users should not have to annotate the code that they're checking; on the other, they should not have to hack directly on the solver's internal representation of constraints. Even turning an LLVM variable into a solver's internal representation—a fixed-width vector of bits called a bitvector—is complicated: if the variable is a struct, is it padded, and if so, how much padding goes between each element?

2. **Expressive**: we can't anticipate all the extensions and checkers that Sys clients may want, so our challenge is to ensure that they can express any checkable property or take advantage of any latent program fact. We arrived at two rules for ensuring that clients of extensible systems can express things that their designers did not anticipate. First, to make sure that clients can express anything computable, they must be able to write Turing-complete code. Second, to make sure that their interface to the system internals—in this case, the static extension and symbolic checkers' interface to Sys internals—is sufficiently powerful, core components of the system *itself* must be built atop the same interface. In contrast, many checking systems have a special, internal interface that built-in checkers use, and a bolted-on, external interface for "extensions." Invariably, the extension interface lacks power that the internal interface has.

3. **Simple**: it should be possible to iterate quickly not only on checkers but also on components of the core system—and changing 6,000 lines of code is easier than changing 60,000. This is especially important for symbolic checking tools because they are inherently complex, built from tightly-coupled, large subsystems, and often operate in feedback loops where each symbolic bit is the child of thousands of low-level decisions. A mistake in a single bit can cause an explosion of false reports that are hard to understand and hard to fix; mistakes that lead to false negatives are hard to find at all.

4. **(Type) Safe**: debugging symbolic execution errors can be nightmarish, since fifty constraints can define a single variable that has a single incorrect bit. We want a system that makes it as easy as possible to get constraints right, and types can help us avoid malformed constraints early.

In the rest of this section, we quickly describe the design of the static extension system. Then, we describe the challenges of building symbolic checkers, and how SysDSL addresses those challenges by fulfilling our design principles.

3.1 Static extensions

Building extensible static checking systems is already the focus of significant work in both academia and industry [27, 39, 60, 62, 65, 70, 84, 87, 122]. Since the details of our static extension system are relatively standard, we only discuss one idiosyncrasy of Sys's static system here: Sys does both its static and symbolic passes on LLVM IR (or bytecode). Typically, static tools want to check the highest-level—most semantics-preserving—representation possible, because the more information they have, the easier it is to find errors and suppress false positives. For example, running checkers for the C language after the preprocessor can cause challenges, since checkers don't know that, say, 127 is actually `MAXBUF` or that a strange chunk of code is actually a macro. Running checkers on bytecode is even more suboptimal in some ways, but we do it because: (1) it makes communication between the static and symbolic passes simple; (2) we can check any language that emits LLVM IR; (3) it lets us "see inside" complicated C++ code for free; and (4) it allows our checkers to comprehend and take advantage of compiler optimizations (§6).

3.2 Specifying symbolic constraints is hard

Users generate their own constraints differently depending on which symex system they use: some systems require language-level annotations, while others have users hack almost directly on SMT constraints. We decided to build SysDSL because of our experience building and using both kinds of tools, which we describe below.

KLEE users express invariants by providing C annotations like "a Bignum's negative field must be either one or zero." According to the main UC-KLEE implementer, David Ramos, naively written annotations would cause KLEE to spin forever—in effect, the annotations would generate LLVM IR that was adversarial to the tool. To write useful annotations, users needed to understand what LLVM IR the C compiler would generate, *and* understand whether or not that IR was compatible with KLEE. For example, David avoided C code that would generate certain LLVM "or" statements, since these statements triggered excessive KLEE forking. David's and our own experiences with KLEE convinced us that we needed a high-level way of expressing constraints that didn't force users to emulate a C compiler.

At the same time, checking LLVM IR by hacking directly on SMT constraints—as we did in early versions of Sys—had its own challenges. LLVM IR and SMT solvers have different basic types (e.g., rich structs vs. simple bitvectors) and different correctness requirements. As an example of the latter, the Boolector SMT solver's [107] logical left shift operator required the width of the second operand to be *log*2 of the width of the first operand; at the IR level, there is no such restriction. Thus, in the middle of trying to write a checker, we would forget the SMT requirement, use the shift, hard crash the solver, add some width castings, get them wrong, etc. In addition to accounting for SMT requirements like left shift, our old approach required users to manually account for LLVM's requirements (e.g., by correctly padding their own structs). We ran into similar problems using `angr` [131] (e.g., solver crashes due to adding variables of incompatible bitwidth), but with the addition of Python dynamic type errors. After that, we wanted to express constraints in a way that protected users from hand-translating IR into SMT.

3.3 Our solution: SysDSL

Sys clients use the SysDSL to write symbolic checkers like the `malloc` checker in Section 2 (Figure 5). The DSL exposes simple, safe LLVM-style operations that it automatically translates into Boolector SMT bitvector representations [43, 107]. In particular, with SysDSL, users can create symbolic variables and constants from LLVM ones; perform binary operations, assignments, comparisons, and casts on these variables and constants; set and get fields in symbolic aggregate structures; and, load and store to symbolic memory. We also provide a library with symbolic operations like `memcpy` that builds on top of the core primitives.

```
1  translateAtomicrmw result rmwOp addr val = do
2    -- Get symbolic variable for LLVM operand addr
3    addrSym <- getOperand addr
4    valSym <- getOperand val
5    -- Get the LLVM operand val's type
6    let operandType = typeOf val
7    -- Load value stored at symbolic addr in symbolic memory
8    oldValSym <- load addrSym operandType
9    -- Do the symbolic rmw operation with two symbolic vars
10   newValSym <- rmwOp oldValSym valSym
11   -- Store the symbolic result to symbolic memory at addr
12   store addrSym newValSym operandType
13   -- Assign the symbolic old value to be the result
14   assign result oldValSym
```

Figure 6: Implementation of the `translateAtomicrmw` LLVM instruction in SysDSL.

Though SysDSL is designed for writing checkers, we also used it to implement every LLVM instruction that the symbolic engine supports, guaranteeing that it's powerful enough to express whatever users want. As an example, we walk through our implementation of an LLVM IR instruction to show how the DSL works. The `atomicrmw` instruction in Figure 6 atomically updates memory using a given instruction (e.g., addition). Given address `addr` and value `val`, the LLVM `atomicrmw` instruction: (1) reads the value, `oldValSym`, at address `addr`; (2) performs the given operation (e.g., addition) with `oldValSym` and `val`; (3) writes the result back to `addr`; (4) returns `oldValSym`.

First, and most importantly, SysDSL eliminates a whole class of type and logic bugs that arise from operating on raw SMT bitvectors. For example, if `oldValSym` and `valSym` (line 10) have different bitwidths, the SysDSL will exit with an informative error. It also prevents more subtle type errors: it lets us ignore the fact that `addr` would be a 32- or 64-bit pointer, and that memory could be an array with blocks of any size. If, say, `addr` is 32-bits in an LLVM file that specifies 64-bit pointers, the SysDSL will exit with an error.

Second, SysDSL exposes functions that are polymorphic over LLVM types to reflect LLVM's polymorphism—e.g., that `rmwOp` (line 10) operates on all widths of integer and vectors—and to simplify both the symex engine and checker implementations. For example, `val` could be a vector or a scalar of any width. Internally, the SysDSL handles the operation accordingly—e.g., for vector `vals` it will automatically decompose the vectors, un-pad the elements if they are padded, add each pair of elements, re-pad the result, and re-assemble the result vector. Doing this manually is both cumbersome and error-prone.

SysDSL also automatically manages variable bindings, mapping an LLVM variable to its corresponding SMT variable. For example, the `getOperand` DSL function on line three takes an LLVM operand as input and returns the symbolic SMT bitvector representing that operand. Internally, this function creates a new bitvector for the LLVM operand if one has not already been created, and returns the existing bitvector otherwise. Similarly, `load` and `store` always load from and store to the most recent version of symbolic memory. Even this seemingly simple task is error-prone when using SMT libraries directly (since users must manually model scope, loops, etc.).

Finally, SysDSL does not bind users' hands: they can compose existing operations to create their own custom operations; the `atomicrmw` LLVM instruction is one example of how to compose new instructions out of SysDSL functions. If, for some reason, users *want* direct access to our Boolector SMT bindings, they can import them; since DSL and bindings functions operate on the same constraint representation, they can interoperate, too.

4 Memory design

Because memory modeling is one of the hardest parts of symbolic checking, this section discusses how Sys models memory. We use KLEE as a comparison point, since it: (1) also focuses on bit-precise symbolic execution and (2) is relatively well known [47] (e.g., it has its own workshop [23]).

Memory In order to perform queries on a memory location in the checked program, a symbolic tool must map program memory to a corresponding memory representation in its constraint solver. The most natural approach (and what Sys does) is to represent memory in the same way as most modern hardware: as a single, flat array.

In contrast, KLEE (and UC-KLEE) represents each object as its own distinct, disjoint symbolic array (you can view this as segmentation). This is because manually segregating arrays lets the solver avoid reasoning about all reads and writes at once; when KLEE was created, solvers had less sophisticated optimization heuristics for arrays, so separate arrays were essential for performance. If a pointer dereference `*p == 0` can point to N distinct symbolic objects, KLEE uses the constraint solver to resolve each option, and fork the current path N times to explore each one separately. This is because KLEE's solver requires that constraints refer to arrays by name, i.e., constraints cannot use "pointers" to arrays.

Sys can use a single flat array for two reasons. First, modern constraint solvers have much better support for arrays, and second, Sys's *much* smaller window size means that there are simply many fewer memory accesses to handle. With a single flat array, every object's address becomes an integer offset from the base of the symbolic array. These offsets can be concrete values or—crucially—fully symbolic expressions. If we use array *mem* to represent memory and p to be a fully symbolic expression, the query `*p == 0` directly translates to $mem[p] = 0$. By using flat memory, Sys sidesteps enumerating all of a pointer's pointees—the SMT solver takes care of that.

A single flat memory array makes translating code to constraints simple. Double-, triple-, quadruple- (or more) indirect pointers take no special effort; `***p == 0` simply becomes $mem[mem[mem[p]]] = 0$. Dereferences work naturally even if naughty code casts pointers to integers and vice versa, or mutilates them through combinations of bit-level hacks. In contrast, just for double indirection, KLEE requires multiple levels of forking resolution.

Shadow memory Flat memory also makes checking easier. Checking tools often need to associate metadata with memory locations. Does a location contain a pointer? Is it uninitialized? Is it deallocated? The wrong way to track this information, for both dynamic and symbolic tools, is by using a special "canary" value [82]. If checked code ever stores the canary bit-pattern itself, the tool will flag false positives, and tracking small units like single bits is clearly infeasible. The problem gets worse for underconstrained symbolic tools. Consider an uninitialized memory checker that stores a canary bit-pattern to all uninitialized pointers. This checker cannot do queries asking if pointers *may* be uninitialized, since if pointer `p` is initialized to point to fully-symbolic `v`, `v` can equal the canary. Instead, the checker asks if pointers *must* be uninitialized. This restriction goes a long way to defeating the point of symbolic checking, since (among other issues), the checker will miss all errors where a pointer could point to both initialized and uninitialized locations.

The standard approach that dynamic tools like Valgrind [106], Purify [82], and Eraser [123] take is to associate each memory location m with a corresponding *shadow memory* location m' that stores metadata about m. They can track even the state of a single bit by setting its shadow location to an integer value corresponding to "allocated," "freed," or "initialized." To the best of our knowledge, UC-KLEE is the only symbolic tool with shadow memory, and it was a 5–10KLOC effort that no tool (that we know of) has since replicated.

Sys implements shadow memory as well—easily, in twenty lines and an afternoon, because it represents memory as a single flat array. Shadow memory is separate, configurable array. As a result, queries on shadow memory are almost direct copies of queries on memory, perhaps with a scaling adjustment. For example, if the user tracks a shadow bit for each location, the expression `*p` maps to $mem[p]$, and the expression $shadow[p/32]$ checks `p`'s shadow bit (assuming 32-bit pointers).

Drawbacks The flat memory model has a number of drawbacks, though: first, it may be too slow for large window sizes and full-program symbolic execution. Second, in a flat memory model, out-of-bounds memory accesses turn into out-of-bounds accesses in symbolic memory. This means that any memory corruption in the analyzed program becomes a memory corruption in the analysis. This could be fixed by tracking a base and bound of each object in shadow memory, and then preventing—but reporting—out-of-bounds accesses.

5 Using Sys to find bugs

In this section, we evaluate Sys's:

1. **Expressiveness:** can we use the SysDSL to express real, diverse checkers and suppression heuristics?

2. **Effectiveness:** can we use Sys to find new security bugs in aggressively tested, huge codebases without sieving through thousands of false positives?

We answer these questions by implementing three checkers that look for two kinds of classic memory safety bugs—use of uninitialized memory and out-of-bounds reads and writes—in browser code, and one system-specific checker that finds unvalidated use of untrusted user data in the FreeBSD kernel.

Workflow We built and debugged checkers on parts of browser code (e.g., the Prio or Skia library) on our laptops. For entire codebases, we ran Sys on a large machine: Intel Xeon Platinum 8160 (96 threads) with 1TB of RAM, running Arch Linux (2/22/19). We check Firefox changeset:commithash `503352:8a6afcb74cd9`, Chrome commit `0163ca1bd8da`, and FreeBSD version 12.0-release. We configured the checkers to run quickly enough that we could debug problems easily: the uninitialized checker uses a bound of 5 blocks, the out-of-bounds 15, and user input 20; we set the solver timeout to 5 minutes. Chrome took longest (under an hour for the out-of-bounds checkers and six hours for the uninitialized memory checker) while FreeBSD was quick (six minutes for user input). All symbolic checkers reject $\geq$98% of statically proposed paths. We discuss block bounds and timeouts further in Section 7.

Bug counting We only count unique bugs: if multiple reports share the same root cause (e.g., an inlined function), we only count a single bug. If the same bug occurs in both browsers (e.g., [5]), we only count it once in the total tally. We mark bugs as unknown if we were unable to map their LLVM IR error message back to source (e.g., because of complicated C++ inlining).

How good is the code we check? The main systems we check—Chrome and Firefox—are some of the most aggressively checked open-source codebases in the world. Both browsers run bug bounty programs that reward security bug reports [51, 103]. Mozilla's program has paid over a million dollars since 2004 [103], and Chrome's most common bugs yield \$500 – \$15,000 [51].

Google runs a massive distributed fuzzer on Chrome 24/7 using over 25,000 machines [21] using three different dynamic sanitizers: AddressSanitizer (ASan) [55] (e.g., for buffer overflows); MemorySanitizer (MSan) [56] (e.g., for uninitialized memory); and UndefinedBehaviorSanitizer (UBSan) [57]. Chrome also encourages developers to write fuzz targets for the their own components [102], and combined, Google fuzzers and test cases reach 73% line coverage of the entire browser [54]. Firefox has a whole team devoted to fuzzing [125], and their JavaScript engine alone ran six different fuzzers as of 2017 [88]. They direct developers to use sanitizers [24] and Valgrind [106], and recently rolled out the ASan Nightly project, where regular users browse the web with ASan enabled—any error triggers an automatic bug report, and any cash bounties are awarded to the user [67].

The browsers also use static tools. Chrome recommends that developers run Clang's core, C++, "Unix", and dead code checkers [52]. Firefox automatically runs static checkers on every submitted patch [104]. These include: (1) Mozilla-specific checkers; (2) Clang-tidy lints; and (3) traditional Clang static checkers. Firefox also runs the Infer static analyzer [15] alongside their Coverity scans (integrated in 2006) [20], which resulted in many thousands of bug fixes.

How good are the bugs we find? Our checkers focus on low-level errors like uninitialized memory and buffer overflows because these are the same bugs that almost every tool we mention in this section detects—so finding these bugs is a better test for Sys than finding errors that other tools have never tried to find. The bugs are also not just new introductions to the codebase. We looked at how long each bountied bug existed, because those seem like the ones other people are most incentivized to find. The Prio bugs have existed since Prio's introduction last year (§5.1), the SQLite pattern has existed for at least nine years (§5.2), and the Opus codec bug has existed for three and a half years (§5.3).

5.1 Uninitialized memory

This section describes our uninitialized memory checker. We start with this error type because it is arguably the most heavily picked-over of any bug type (more even than buffer overflows). The results in Figure 7 show that Sys is effective—it finds 21 errors—and we describe how the checker works and its results below.

5.1.1 How the checker works

Static extension: a simple, somewhat conservative pass that marks potential uses of uninitialized stack variables. For each stack allocation s, the extension performs a flow-sensitive pass over all subsequent paths. If there is no obvious store to s, the extension marks the first load of s as potentially uninitialized. The extension does not track pointer offsets, instead considering every new offset as a new tracked location.
Symbolic checker: uses Sys's shadow memory (§ 4) to detect uses of uninitialized memory, similar to concrete tools like Valgrind [106] and Purify [82]—with the advantage that it can reason about many possible locations at once (e.g., all locations that a symbolic pointer or a symbolic array index could refer to).

Sys runs the checker symbolically on each path flagged by the static pass. The start of each checked path is a stack allocation s that is potentially used uninitialized. The checker associates each bit in s with a shadow bit s_b and initially sets each shadow bit s_b to 1 (uninit). At each store, it writes all associated shadow bits s_b to 0 (not-uninit). Finally, at the end of the first block in which s is read, the checker runs the following snippet with s as `uninitVar`; it will emit an error if any bit in s_b is set:

```
uninitCheck uninitVar uninitType = do
  uninitSym    <- getName uninitVar
  shadowResult <- loadShadow uninitSym uninitType
  isSet        <- uninitConst uninitType
  assert $ isEq uninitType shadowResult isSet
```

The last line adds a solver constraint that the checked location `uninitSym`'s shadow memory, `shadowResult`, has a bit set (implying uninitialized). The solver runs immediately afterwards and "stores" its result—SAT implies a read of uninitialized memory—so that the checker can use it (e.g., to report an error to the client). Note: the checker is not doing a concrete check of a single value. The loaded location, `uninitSym` and thus `shadowResult` can be symbolic expressions that refer to many storage locations simultaneously. The solver will determine if any value on the checked path could cause any of these locations to read even a single uninitialized bit.

False positives: perhaps because this checker examines so many locations, it was the most sensitive to false positives caused by impossible values. We discuss the two most interesting sources of false positives below.

First, for speed, the checker does not enter functions and instead takes advantage of Clang's inlining. This initially caused a serious number of false positives whenever an uninitialized variable `x` was passed to a skipped function and then used (e.g., `init(x); *x;`). Before we built shadow memory, we tried two failed approaches:

1. Using static analysis to determine at the IR level which pointers were passed to skipped function calls. This was brittle at deciphering the casts, loads, and pointer offset calculations that created escaping pointers.

2. Suppressing the problem symbolically, by storing unconstrained values to each pointer passed to a function. This introduced more correctness problems: LLVM IR uses pass-by-reference whenever it can, so the checker ended up modifying almost all values passed to functions.

The solution using shadow memory is more robust: for each pointer argument, add a constraint that uses the argument expression to exactly describe which shadow locations to clear (e.g., most simply, for unentered `foo(p)`, clear every bit in `p`'s shadow). The exact constraint meant we did not need to manually determine which pointers were passed to functions, and shadow memory let us avoid canaries. Unintuitively (to us), this version of *symbolic* false positive suppression was actually far easier and more effective than the static one.

The second initial source of false positives arose because we run checkers on the optimized release builds of Chrome and Firefox, the code that gets shipped to millions of users. Optimizations shrink the IR by almost an order of magnitude (good) but also strip `ASSERT`s (bad). Both browsers rely on `ASSERT` to express invariants, which means checking optimized code can yield false positives:

System	True	False	Unknown
Chrome	6	5	1
Firefox	12	3	3
Total Browser	13	7	3
FreeBSD	8	2	0
Total	21	9	3

Figure 7: True bugs and false positives for the uninitialized memory checker. All true browser bugs are reported and confirmed by a triage developer with two exceptions, where we reported directly to the library maintainers and have not heard back. FreeBSD has not responded.

```
ASSERT(num > 0);
ASSERT(num >= j);
for (int i = 0; i < num; i++) x[i] = i;
return x[j]; // j is unsigned
```

Without the `ASSERT`s, the checker does not know that `j` is less than `num`, and thus that the final `x[j]` can never be uninitialized. To avoid this, we re-run all buggy function snippets on the debug version of the browser; there, the tool can tell that the "buggy" path is actually infeasible.

5.1.2 Checker results

Figure 7 breaks down the 21 bugs this checker found. Three Firefox bugs were marked sec-medium (equivalent to bugs that, say, can be used to leak cookies); together, these bugs were rewarded a bounty and assigned a new CVE. A Chrome bug also received a CVE. The false positive rate for this checker is relatively high, but we hope to improve it by jumping back to callsites. Below, we discuss two cases where Sys found bugs that other tools missed.

A benefit of checking IR: checking IR means that we see any compiler-generated code, and thus can detect errors in it, or errors in assumptions programs make about it. For complex languages like C++, doing this reasoning with a high-level checker can be hard, since it is not always obvious what the compiler might do—or even that it will do anything at all.

The following uninitialized memory CVE in Chrome's WebRTC module [7] is a good example. Here, a compiler-generated default constructor never sets a field that a cleanup function uses:

```
/* third_party/webrtc/modules/audio_processing
↪  /aec/echo_cancellation.cc */

Aec* aecpc = new Aec();
...
aecpc->aec = WebRtcAec_CreateAec(aecpc->instance_count);
if (!aecpc->aec) {
  WebRtcAec_Free(aecpc);
```

The `Aec` constructor is defined with C++ 11's `default` keyword. This compiler-generated constructor (not shown) does not initialize the `far_pre_buf` field of the `aecpc` object; instead, Chrome relies on code to call `WebRtcAec_Init` to initialize the object. Unfortunately, when the allocation function (`WebRtcAec_CreateAec`) returns null, this field remains uninitialized and *is* used by `WebRtcAec_Free`.

Checking checkers: checkers have errors, just like the code they check. Errors that lead to false negatives are especially pernicious because they are silent. For example, this uninitialized memory bug from Firefox's Prio library for privacy-preserving data aggregation [83] should have been caught by Firefox's regular Clang checks:[6]

```
/* third_party/prio/prio/serial.c */

static SECStatus
serial_read_mp_array(msgpack_unpacker* upk, ...,
                     const mp_int* max)
{
  SECStatus rv = SECSuccess;
  P_CHECKCB(upk != NULL);
...
  msgpack_unpacked res;
...
cleanup:
  msgpack_unpacked_destroy(&res);
```

Here, `P_CHECKCB` checks that `upk` is null and, if so, goes to `cleanup`. `cleanup` uses the `msg_unpacked_destroy` function to free fields of `res`—but `res` hasn't been declared on this path, let alone initialized. Given that this bug was serious enough to lead to a bounty and CVE, *missing* it may also be a serious bug in itself. Running multiple tools is a way to find such mistakes, but similar tools can have similar mistakes. Since Sys is very different from most industry tools, it should be better able to expose their false negatives (and vice versa).

In practice, browser developers really do update their checking tools in response to bug reports. After looking an NSS bug Sys found [12] (and an audit of NSS for more occurrences of the bug), a triage developer said "at the very minimum, the problem in `PRZoneCalloc` should be found by something. If not, we have static analysis problems." They changed their Coverity configuration so that it would find the missed bug.

5.2 Heap out-of-bounds

After uninitialized memory bugs, stack and heap buffer overflows may be the second most widely-checked defect in the codebases we examine. Overflow checking is popular because overflows are the most common way for attackers to hijack control flow—stack buffer overflows are used to overwrite return addresses, while heap buffer overflows are used to overwrite function pointers and virtual table pointers [64, 139].

This checker (Figure 8) discovered 21 out-of-bounds bugs, including a group of 13 in Chrome's SQLite with a bounty and a CVE. It also discovered a CVE in Firefox. Our guess for why Sys found so many errors is because this check requires both complicated reasoning (hard for static) and edge case values to trigger problems with bit-widths and integer wrapping (hard for static, dynamic, and humans). Since Section 2 already described this checker, we now mention one difference in how it makes up fake state, and then discuss results.

[6]The Prio author runs his own Clang scans that also missed the bug [58].

All other checkers use Sys's default strategy of allowing unknown integer values to be anything, but this checker makes one change to reduce false positives. Many system components we check have an internal security model where values from outside (e.g., user SQL) need to be checked, but data internal to the browser is trusted. The checker approximates this split by assuming that any value coming from inside a data structure has already been checked to be "small." It assumes all other values can be anything. Without this trick, the checker was unusable; with it, the results were clean and serious. This example shows the power of extensions. Because they are flexible, we can use them to implement *programmatic annotators*: rather than manually, laboriously marking each field as safe, we use a few lines of code to mark them all.

Section 2 presented the most serious bug this checker found. As a twist on Section 5.1, where we discussed using Sys to improve other tools (and vice versa), Sys discovered—from first principles—a pattern that much simpler tools can express and check. One of these bugs is in Chrome's LibDRM, an interface for communicating with GPUs [6]:

```
/* third_party/libdrm/src/xf86drmMode.c */

new->items = drmMalloc(old->size_items *
  ↪ sizeof(*new->items));
...
memcpy(new->items, old->items, old->size_items *
  ↪ sizeof(*new->items));
```

This code looks fine at first glance: both the `memcpy` size and the allocation size of `new->items` are exactly the same. But `drmMalloc` takes a 32-bit `int` input, while `memcpy` takes a 64-bit `size_t`. For realistic values, `size_items` can be large enough to wrap a 32-bit integer but not a 64-bit integer: the size passed to `drmMalloc` will wrap around to a small value and become the target of huge overflow when `memcpy` copies the unwrapped number of bytes. We found three separate instances of `malloc` routines designed to take `int`s (or `i32`s on x86-64) used with memory operations designed to take `size_t`s (or `i64`s on x86-64). Using these bugs as examples, a simple static checker should be able to find this pattern, too.

In response to our reports, the LibDRM team is fixing their allocation routine to take a `size_t` [6]; the main SQLite author patched their code to "use 64-bit allocation routines wherever possible," according to his commit message [1]; and a security lead at Firefox asked for an audit of the allocation routines in NSPR and NSS [68, 69], network runtime and TLS code that uses small mallocs [12].

5.3 Concrete out-of-bounds

This section focuses on a specialty of static checkers and even compilers: stack and heap out-of-bounds bugs caused by indices that are *always* concretely out-of-bounds. There should be almost none left in the code we check. Surprisingly, out of the four reports we've examined so far, Sys found three confirmed bugs (with one false positive), including a bountied bug (Figure 9) in Chrome's audio muxer. According to the bug

System	True	False	Unknown
Chrome	19	3	2
Firefox	16	6	3
FreeBSD	0	0	1
Total	21	7	4

Figure 8: True bugs and false positives for the out-of-bounds checker. We have reported all true bugs and they have been confirmed by at least a triage developer. We run on `O1` for this one checker, since duplicate reports from inlining make production builds overwhelming.

report, fuzzers missed one of the bugs because the incorrect access was still within the bounds of the object [16].

Static extension: tags three actions:

1. Concrete `phi` nodes (e.g., `phi i32 [5, %label]`), which choose between values flowing into a basic block, and are one way of loading constants into operands [95].

2. Compiler-generated `undef` constants [96], used to denote undefined values (e.g., the result of an undefined operation). Since `undef` is a value that allows any bit-pattern, using it as an index may overflow.

3. Any `getelementptr`, LLVM's offset calculation instruction [94], with a concrete index.

For efficiency, the static pass does a simple analysis to determine which constant values tagged by the first two cases could reach the third (array index), and passes this information to the symbolic checker. The static pass currently ignores indices into: parent class objects, since these objects may have a different layout than child object; dynamically-sized struct fields (i.e., in C++ accesses off the end of arrays of size `[1 x type]` in structs); single-index out-of-bounds (because of C++ iterators); and union types. We tried to write a simple checker, but a smarter checker will likely yield more results.

Symbolic checker: determines that the out-of-bounds indexing is *possible*. Since we are checking a purely concrete property, and in contrast to the other checkers, this symbolic pass just uses Sys to prune false paths.

5.4 Unvalidated user data

Many symbolic tools can't handle operating systems code, but Sys handles it as easily as anything else: simply jump to the code and check it. As a quick proof of concept, we wrote a checker for FreeBSD, found two confirmed bugs (no false positives), and stopped [25, 26]. This also shows that Sys makes it easy to check system-specific properties.

For space reasons we give only a brief summary. The checker traces untrusted values copied from user space, using the solver to flag errors if (1) an untrusted value used as an array index can be enormous; or (2) an untrusted value passed as a size parameter (e.g., to `memcpy`) could cause overflow.

6 Evaluation

In this section, we experimentally compare Sys with state-of-the-art static analysis and symbolic execution tools (§6.1). We then describe our experience and the experience of others writing checkers and using Sys to find bugs (§6.2). Finally, we evaluate the impact of several Sys design choices (§6.3).

```
    /* src/media/muxers/webm_muxer.cc */
    /* Tiny <opus_header> passed to WriteOpusHeader */
303 uint8_t opus_header[OPUS_EXTRADATA_SIZE];
304 WriteOpusHeader(params, opus_header);
    /* WriteOpusHeader writes past <opus_header> */
 20 void WriteOpusHeader(const media::AudioParameters&
    ↪  params, uint8_t* header) {
...
 41   if (params.channels() > 2) {
...
 48     header[OPUS_EXTRADATA_NUM_COUPLED_OFFSET] = 0;
 49     // Set the actual stream map.
 50     for (int i = 0; i < params.channels(); ++i) {
 51       header[OPUS_EXTRADATA_STREAM_MAP_OFFSET + i] =
 52         kOpusVorbisChannelMap[params.channels() - 1][i];
 53     }
 54   }
...
 48 }
```

Figure 9: Bountied, medium-severity bug in Chrome [2]. The array `opus_header` is allocated with `OPUS_EXTRADATA_SIZE` elements, which is 19. Then, `opus_header` is passed to `WriteOpusHeader`, which writes out-of-bounds of `opus_header`: the writes on lines 48 and 51 are to index 20 and 21 respectively.

Tool	True	False	Total
Clang	13	108	121
Sys	12	3	18 (3 unknown)
Semmle	2	58	60

Figure 10: Uninitialized memory bugs that each tool found in Firefox.

6.1 Comparing Sys's approach

To understand the importance of our extensible, combined static and symbolic approach for checking large codebases, we run Sys and a variety of other tools on the Firefox web browser. We use Firefox instead of Chrome because Chrome has an intricate build system, one that is hard to interface with many tools (e.g., Chrome downloads its own version of its compiler). Overall, we find that Sys works well on larger code compared to other symbolic execution tools, and it has a lower false positive rate than standard static analysis tools. We do not check smaller codebases; existing symbolic tools can handle this task and users should continue using them.

6.1.1 How does Sys compare to static approaches?

To understand the effect of symbolic reasoning, we compare Sys to two state-of-the-art static analysis tools—the Clang Static Analyzer [87] and Semmle [126]—on finding uses of uninitialized memory. We choose these two analysis tools because (1) they scale to huge codebases like Firefox and (2) Mozilla already uses both tools (e.g., they are the two systems in their new static analysis bug bounty program [119]); there are many other similar tools [19, 48, 85]. We evaluate the tools on uninitialized memory bugs because both tools have built-in checkers for this bug class—and, for example, Mozilla already runs the Clang checkers daily.

Clang Clang has six built-in checkers that can identify uninitialized memory bugs: “assigned value is garbage or undefined,” “branch condition evaluates to a garbage value,” “undefined pointer value,” “garbage return value,” “result of operation is garbage or undefined,” and “uninitialized argument value.” We ran the six checkers on Firefox; together they flagged 371 potential bugs. We manually examined each report to determine: (1) if the report was caused by purported stack uninitialized memory or by something else (e.g., shift by a negative number) and (2) for the stack uninit reports, whether the result was a true positive or a false positive. Table 10 summarizes our findings: of the unique stack uninit flags, Clang found 13 true bugs with 108 false positives. This contradicts our original hypothesis that few statically-detectable bugs still exist in browser codebases; instead, we found that many of the bugs Clang detected were still unfixed because of the large number of false positives the tool produced; examining 371 reports for 13 true uninit bugs is daunting.[7]

We marked a bug as a false positive either because the bug was impossible to reach, i.e., there was no feasible path to the uninit use, or because the variable was actually initialized before use. For bugs we were not not completely confident in, we checked the latest Firefox source for the bug and checked whether or not the alert had been suppressed by Firefox: if the bug was still in the source but not in the latest Clang reports, we marked it as a false positive (since it had likely been suppressed). We also checked the latest Firefox source and Clang report for bugs we were confident to be true positives. Of these, eight bugs were either fixed or removed from the codebase. The other six bugs we marked as true positives had disappeared from reports, i.e., they were (likely accidentally) suppressed, or some heuristic changed, causing them to disappear. We reported these bugs to Mozilla, where four have been confirmed and fixed [17]. In line with previous work [39], this shows how false positives can turn into false negatives: if no one is motivated to go through hundreds of mostly false reports, bugs that a tool finds will never get fixed.

Semmle We also ran Semmle’s default stack uninitialized memory checker—the `cpp/uninitialized-local` query—on Firefox commit `cbd75df`.[8] The checker flagged 465 possible errors, of which we examined the first 60 alerts. We did not inspect all the alerts since Semmle requires source modifications to suppress false positives (as opposed to checker modifications). Sorting the alerts differently did not change the list of bugs and, unfortunately, we could not select a random sample—the Semmle interface is paginated and presents a handful of bugs at a time.

[7]To be fair, some of the 371 reports were duplicates, but Clang does not automatically de-duplicate reports.

[8]The Semmle console we had access to only checked this version.

As with Clang reports, we marked a bug as a false positive either because the bug was impossible to reach or because the variable was actually initialized before use. Of the 60 flagged bugs, two were true positives. Since they were on the same line (for two different variables), we filed a single, now confirmed and fixed, bug [18].

Reasons for false positives In our analysis of the Clang and Semmle reports, we found that almost all the false positives were because these static tools do not reason about values. For example, Semmle flags the `color` variable on line 157 of:

```
/* dom/html/HTMLHRElement.cpp */
59  nscolor color;
60  bool colorIsSet = colorValue &&
      ↪  colorValue->GetColorValue(color);
...
156 if (colorIsSet) {
157   aDecls.SetColorValueIfUnset(eCSSProperty_color, color);
158 }
```

The variable, however, is conditionally initialized on line 60 (in `GetColorValue`) and only used on 157 if the condition is true and the initialization routine succeeded. Extending Clang and Semmle with basic value reasoning can eliminate simple false positives like this example, but many of the bugs we analyzed were more complex—and addressing this problem in general is precisely a symbolic execution task. Alternatively, we could send Sys down all paths that Clang or Semmle identify as possibly buggy.

Reasons for false negatives Sys did not identify the two Semmle bugs or ten of the thirteen Clang bugs. Four were due to unentered function calls; four appeared beyond Sys’s block bound; two bugs were optimized away by the compiler; one looks safe in LLVM IR, so we are waiting for more information from Firefox developers; and one is very difficult to map LLVM IR back to source. Based on these results, we think that it makes sense to (1) enter all function calls uninitialized variables are passed to and (2) optimize Sys so that we can increase its block bound on large codebases.

6.1.2 How does Sys compare to symbolic approaches?

To understand the effect of the static analysis pass, we compare Sys with KLEE and `angr` running in underconstrained mode [45, 131].[9] We use these tools to represent the fully symbolic approach and the UC approach, respectively.

Firefox We ran `angr` in its default configuration (but using underconstrained mode) to detect uninitialized memory in Firefox. It spent roughly twenty-four hours in a profiling function before we stopped it, and it did not detect any errors. We did not run KLEE on Firefox largely because our `angr` experiment: since UC symbolic execution doesn’t scale to the browser, full symbolic execution is even less likely to.

[9]KLEE version 2.0 and `angr` 8.

Checker	Static LOC	Symbolic LOC
Uninit	132	106
Heap OOB	273	62
Concrete OOB	148	14
User Input	135	13
Total	688	195

Figure 11: Lines of code for each checker (commit 26d7c7af). The whole system is 6,042 LOC, not including bindings or the compiler or SMT solver, and the symbolic execution engine is 2,168 LOC.

SQLite We tried to use KLEE on a smaller part of Firefox: the SQLite 3.28 database shared library.[10] We ran KLEE for three days, configured with a symbolic input file of 4096 bytes and symbolic stdin of 1024 bytes; we used a large file because many bugs (e.g., our malloc bug) require very large tables. The tool produced 1,419,187 test cases in three days, none of which exposed errors in SQLite (most yielded malformed database errors or returned the version number of SQLite). KLEE is more likely to do well given a smaller input file or a partially concrete and partially symbolic file.

6.2 Experience writing and using checkers

This section describes our experience building and using Sys and the experience of others using Sys and SysDSL to write checkers and find bugs.

Building and using Sys Although we spent over a year building early versions of Sys, things moved quickly once the system was done: the first author debugged the system, wrote every checker, and validated and reported all bugs in about three months. SysDSL allowed us to experiment with different ways of expressing checked properties and suppressing false positives over that three-month period: recall that Section 5.1 and Section 5.2 give examples of false positive suppressions, while Section 5.4 gives a brief rundown of a system-specific checker for FreeBSD. Each static extension is under 280 lines of code, while each symbolic checker is under 110 (Figure 11). Making a checker typically took a day or two of writing code, running the checker, tweaking the checker, re-running the checker, etc.—and initial results were fast. For example, we found a CVE in a few minutes the first time we ran the uninitialized memory checker (on the Prio library [3]).

The largest time sinks were: (1) writing up bug reports for browser developers to read and (2) coming up with and implementing false positive suppression heuristics. We discussed the latter in Section 5. For the former: Sys automatically indicates the exact line on which the bug appears. Using LLVM's debug information, we determined which line this corresponded to in the browser source code, and tried to figure out if the bug seemed real. Then, for each real bug, we wrote a report explaining that bug, sent it to browser developers, and then communicated with those developers about the details of the report.

[10]We tried `angr` on SQLite, too, but ran into implementation bugs (likely because our use case is not what the tool is actually used for).

Writing checkers To understand the challenges of writing checkers with SysDSL, we report on the experience of the second author of this paper writing their first Sys checker. Their task was to write a checker that could identify simple use-after-free bugs.

The overall effort took three work days, including testing and running the checker on FreeBSD and Firefox. The author used the uninit and user-input checkers as a reference to implement both the static extension and symbolic checker. The static extension tracks freed variables (and their aliases), and flags any uses (operands to load and store, and arguments to function calls). The symbolic checker sets the shadow bits on free, and checks if any shadow bits are set on load, store, and call. The false-positive suppression ignores UAFs in reference counting code.

The final checker (110 LOC extension, 80 LOC symbolic) flagged a true positive bug in Firefox (in the HarfBuzz text shaping engine), which was fixed within a few hours of our report [14]. Sys also flagged a false positive in FreeBSD: a call with a dangling pointer argument where the called function did not dereference the dangling pointer. Since passing dangling pointers across function boundaries is almost always an error, we will report this bug as well.

This qualitative checker-writing experiment revealed two challenges. First, Sys needs utilities to more easily inspect shadow memory; this could have simplified debugging the UAF checker—and any other checkers that rely on shadow memory. Second, Sys needs an interactive (mixed LLVM and source) interface to simplify the task of confirming true positives. We consider these improvements future work.

Using Sys to check other systems Though we explicitly designed Sys to be extensible, existing Sys checkers can be used without modification, too. For example, the program analysis team at a large company used Sys to check their custom operating system, which has been analyzed and tested for seven years since its initial release. The team found three heap out bounds bugs within a week or two of receiving the tool. They also found a bug in our checker—specifically our `calloc` implementation—that was the source of a false positive. Finally, they identified similar challenge to our UAF experiment: some of the checkers' outputs were confusing (e.g., at that time, our checkers had different output formats).

6.3 Micro experiments

In this section, we explore two variables that users control and that can effect checker results: Sys's block bound and the checked optimization level. We ran each of these experiments on Firefox's Prio library, since it contains at least three uninitialized memory bugs, and Sys found these bugs in its default configuration.

Optimization level We ran the uninitialized memory checker on optimization levels `O0`-`O3`, `Os`, and `Oz`, because LLVM IR for the same program looks different across levels.

Sys found no bugs at `O0` or `O1`; one bug at `Oz`; and all three bugs at `O2` (the default level for most of the browser), `O3`, and `Os`. Sys does not find bugs at the lowest optimization levels because its static analysis pass matches on patterns more common in production builds; future work is understanding if Sys can find additional bugs at different, higher optimization levels in the browser, and determining whether building static analysis specifically for lower optimization levels can yield new bugs, as well.[11]

Block bound We ran Sys on Prio (`O2`) with block bounds of 1, 2, 5, 10, 15, 20, and 30. It found three bugs at bounds five and up; at bound two, it found one bug, and at bound one, it found zero bugs. This, in combination with our analysis of Sys's false negatives, suggests that optimizing the system to support longer block bounds is a good first step in increasing the number of bugs Sys is capable of finding. It's possible, though, that longer block bounds will cause more false positives, since more blocks means more opportunities for undefined state to affect the analysis.

7 Limitations and future work

Sys skips code and so is not exhaustive: it doesn't prove the absence of bugs, and may miss bugs because of false positive suppression, solver timeouts, loop bound and offset bound configuration, and the size of the checking window. Other symbolic execution tools like KLEE, UC-KLEE, and `angr` symbolically execute whole programs or whole functions, and so miss fewer bugs but also cannot scale to check browsers as written.[12] Each tool hits a different point on the trade-off curve: on the one end, KLEE is designed for exhaustive checking of small programs, while Sys is meant to incompletely check huge ones (§8). Moreover, `angr` (for example) could implement our scaling strategies, or we could modify Sys to symbolically execute whole functions or programs.

Though Sys has a lower false positive rate than other UC implementations—`angr`'s version has "a false positive rate of 93%, in line with . . . UC-KLEE['s]" [131]—it still produces false reports. Many of its false positives come from unknown caller invariants on the functions it checks. About half of these are obviously false after quick examination; the other half are hard to reason about. In the future, we plan to eliminate the easy half by jumping back to callers and re-checking for bugs.

Sys, like all extensible checking systems (e.g., Pin [98], `angr` [131], Semmle [126], etc.), requires users to write new checkers if they want to find new styles of bugs; users may obviously re-use any existing checkers to find new bugs in different systems. For example, we re-used each checker on each different system without modification. Finally, Sys runs on LLVM IR, which means that developers must be able to compile their code to use it—which can be a problem in practice, for example, when checking closed source systems, or when integrating with a new build system [39, 41].

8 Related work

We designed Sys to check huge (browser) codebases that are thoroughly, continuously, and automatically vetted. To our knowledge, most other symbolic tools check codebases that are orders-of-magnitude smaller than browsers, and most research bug-finding systems in general look at codebases that are less thoroughly checked by state-of-the-art tools. Since many of the challenges we ran into arose precisely because of trying to check very large, very good code, we see our work as largely complimentary to the existing literature.

Flexible symbolic checking. Analysis tools have been using extensions to exploit domain- and program-specific knowledge for many years [62, 90, 98, 135]. Symbolic tools have incorporated these capabilities, but as far as we know, there is no symbolic checking system designed solely for iterative bug-finding. UC-KLEE's main goals were to scale symex while (1) checking C program correctness without user intervention and (2) avoiding false negatives [114]. Though UC-KLEE supports checker extensions, the extensions' false positive rates are high (80-100% for most checkers), and users must specify invariants as C annotations (§3).

Woodpecker [59] *verifies* user-specified rules over complete C programs, so things like false-positive suppression are irrelevant. Woodpecker is built on KLEE and provides four built-in checkers, and it appears that users write checkers that directly manipulate constraints; we discuss Woodpecker more below. Saturn [143, 144] users write checkers by associating finite state machines with program objects. Though Saturn found many locking bugs in the Linux kernel, the tool is not designed to check large C++ codebases (e.g., it relies on a custom front-end compiler and IR that models C, and does not let users encode heuristics or false positive suppressions).

The `angr` [131] framework, originally designed to compare different binary analysis techniques, is used for everything from exploit generation to binary patching. Though we share similar high-level goals with `angr`, they focus on easy implementation of analyses, while we focus specifically on bug checkers—one level of abstraction higher. `angr`'s low-level, untyped interfaces make the tool flexible (e.g., Sys's scaling approach could likely be implemented on top of `angr` to find bugs in binaries). In our experience these low-level interfaces also make it hard to use for bug-finding (e.g., from debugging checkers and heuristics to modifying the tool itself to adding support for multi-threaded execution; §3). Sys, on the other hand, is poorly suited to tasks like reverse engineering.

Combined static and symbolic execution. We are not the first to combine static analysis and symbolic execution. The most relevant work is Woodpecker [59], which significantly speeds up symbolic execution by skipping paths that are not relevant to a given checker. While skipping paths helps, Wood-

[11]For example, it may be able to find undefined behavior bugs that the compiler optimizes away at higher optimization levels.

[12]These symbolic tools can also miss bugs due to small sizes of input objects or their environmental models.

pecker still must find a full path to a bug from `main`. This problem matters less for them, since they check code that is orders of magnitude smaller than browsers.

The Dowser system finds buffer overflow vulnerabilities by combining fuzzing, program analysis, and symbolic execution: it performs static analysis to identify complicated program pieces, and then uses combined symbolic execution and fuzzing to steer the program towards the target lines [81]. Deadline [145] finds double fetch bugs in OS kernels—Linux and FreeBSD—by using static analysis to prune uninteresting paths and focus the symbolic execution to paths that contain multiple reads. Gadelha et al. [71] implement an extension to the Clang Static Analyzer that reduces false positives by encoding the path constraints leading to a bug as SMT constraints; if the constraints are unsatisfiable, it suppresses the bug report (e.g., they find 7% of bugs to be unreachable). Finally, Parvez et al. [109] use static analysis to identify potentially buggy statements, and then use symbolic execution to synthesize test cases that hit the statements.

Other systems combine static and symex for failure reproduction. Zamfir et al. take a bug report and use a combination of static analysis and symbolic execution to reproduce the bug. Chopper [140] users specify uninteresting parts of a program, which the tool then excludes (with static analysis) before performing symbolic execution. Many others [34, 59, 66, 73, 78, 79, 93, 112, 118, 120, 130, 147, 153] similarly combine static analysis and symbolic execution for testing, verification, and bug finding—from memory leaks to use-after-frees to buffer overflows. All of these approaches demonstrate the power of symbolic execution combined with static analysis. However, none use underconstrained symbolic execution, which is how Sys scales to large code.

Incomplete symbolic execution. Our incomplete symbolic execution builds on prior work. UC-KLEE [115], the first system to support underconstrained symbolic execution [63], deals with the problem of undefined state by cross-checking a patched and unpatched function: if the two versions differ beyond the bug fix, UC-KLEE reports an error. As a result, all state is defined explicitly by equivalence. Our work can be seen as a response to UC-KLEE's (and later, `angr`'s) open challenge to reduce the false positive rate of underconstrained symbolic execution of single versions of functions.

Chopper [140] deals with undefined state by avoiding it: it lazily executes any state that the path under analysis requires. Bergan et al. [38], like our work, allows symbolic execution to start at any program point; they, however, tackle the undefined state challenge by using context-specific data-flow analysis to soundly over-approximate the state. In contrast, our symbolic execution strategy has similarities to call-chain-backward symbolic execution [99] and iterative verification [110].

Combined concrete and symbolic execution Symbolic execution tools (e.g., [45, 132, 143, 144]) have been successful at bug finding, test generation, and partial verification. But, since full symbolic execution struggles to scale [36, 47, 131], much past work has focused on tackling this challenge. Most often, modern tools combine symbolic execution with concrete execution; these *concolic execution* tools (e.g., [42, 46, 50, 61, 74–76, 127]) can run long paths in large programs by executing some code concretely. But the set of code paths and values are inexhaustible, and thus even these tools can easily miss errors by not hitting a given path, or not executing it with the right value. Similar problems arise for other bug finding systems (e.g., [32, 33, 44, 49]).

Finally, for more information on the benefits and drawbacks of underconstrained symbolic execution compared to traditional symbolic execution—in other words, information on the impact of skipping code—Ramos [113] directly compares KLEE and UC-KLEE along a number of axes (e.g., scalability, false positives, etc).

Fuzzing and symbolic execution. Fuzzing has identified more bugs in browsers than any other approach [30], but fuzzers have their own scaling challenges. In particular, fuzzers like AFL [152] have a hard time checking *deep code*. In response to this, various systems, including Driller [137], QSYM [150], CAB-Fuzz [86] and several others (e.g., [97, 100, 108]), combined fuzzing with symbolic execution. T-Fuzz [111], for example, scales fuzzing by skipping complex constrains and uses symbolic execution to determine if the bugs flagged bugs are real; it, however, relies on full symbolic execution which does not scale to checking browsers.

Extensible static checking. There are many extensible static frameworks for bug checking [39, 48, 60, 121]. Hallem et al. [80] present one such system, and the Clang Static Analyzer [87] allows users to write their own static checks using an API. Semmle provides a query language for detecting buggy patterns in source code; they, however, require developers to add inline source annotations [126]. Joern provides a query language for finding bugs and "fuzzy" parsing to avoid constructing full program graphs [85]. These efforts are largely complimentary; indeed, an future direction is to combine such source-level static analysis with low-level symbolic execution.

Memory safety bug checkers We are not the first to identify uninitialized memory, buffer overflow, and use-after-free bugs; we chose these classes of bugs *because* they are aggressively checked for and thus good test cases for new tools. Many static tools identify the bug types we look for: Garmany et al. build a static analysis framework for detecting uninitialized accesses in binaries, identifying seven bugs [72], and tools like the Clang Static Analyzer [87], Coverity [19], and Semmle [126] all detect uninitialized memory bugs in source code. We compare to Clang and Semmle in Section 6; these tools and others [31, 35, 65, 80, 89, 117, 138, 146] also detect overflow and use-after-free bugs statically. Finally, Lee et al. provide a thorough overview of undefined behavior—and how to view certain bug types as cases of undefined behavior [92].

Dynamic tools and “sanitizers” [133] can also detect the bug types Sys finds. MSan [136], UBSan [57], and ASan [128] automatically instrument programs to detect uninitialized reads, undefined behavior, and memory and use-after-free errors, respectively; Ye et al. reduce the overhead of MSan on the SPEC2000 benchmark [148]. Valgrind [106] supports the MemCheck tool [129] that warns about memory errors like out-of-bounds access and uninitialized memory.

Mitigating memory safety bugs There is a large body of work on eliminating and mitigating the classes of bugs Sys checks for. For example, DangSan [141], DangNull [91], and FreeSentry [149] can mitigate use-after-frees; BaggyBounds [28] and others (we refer the reader to [139]) can mitigate buffer overflows; and systems like SafeInit [101] can mitigate uninitialized memory bugs. In practice, browsers rely on sandboxing to contain the damage caused by these classes of bugs, and more recently, they have turned to verification [154] and memory safe languages like Rust [29, 77].

9 Conclusion

This paper presents Sys, an extensible framework for automatically detecting bugs using a combination of static analysis and symbolic execution: static analysis identifies potential errorsites cheaply, while symbolic execution reasons deeply about whether the sites are actually in error. Developers can use existing Sys checkers for uninitialized memory, overflow, and use-after-free bugs, or they can write their own checkers for custom properties. Sys identifies 51 bugs (four CVEs and three groups of bounties) in browsers and operating systems.

Acknowledgments

We thank the reviewers, and our shepherd Thorsten Holz for his insightful comments and help navigating this process. Many thanks to Ranjit Jhala for his always impeccable guidance. Thanks to Craig Disselkoen for work on an early version of the tool, and Diana Young, Mike Walfish, David Ramos, Riad S. Wahby, Andres Nötzli, and Henry Corrigan-Gibbs for their assistance with both prose and ideas. Thanks to everyone who responded to our bug reports for Firefox, Chrome, and FreeBSD, especially Daniel Veditz and Nicholas Nethercote at Mozilla, and Ed Maste, Gordon Tetlow, and Ali Mashtizadeh with FreeBSD. Thanks to Tom Ritter for helping us run Semmle and Evan Johnson for helping us run `angr`. Thanks to Mary Jane Swenson for everything. This work was supported in part by a gift from Cisco, the NSF under Grant Number CCF-1918573 and CPS-1931750, and the Global Research Outreach program of Samsung Research.

References

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=952406.

[2] https://bugs.chromium.org/p/chromium/issues/detail?id=930035.

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1521360.

[4] https://bugzilla.mozilla.org/show_bug.cgi?id=1544181.

[5] https://bugzilla.mozilla.org/show_bug.cgi?id=923799.

[6] https://bugs.chromium.org/p/chromium/issues/detail?id=940323.

[7] https://bugs.chromium.org/p/chromium/issues/detail?id=922882.

[8] https://bugs.chromium.org/p/chromium/issues/detail?id=943345.

[9] https://bugzilla.mozilla.org/show_bug.cgi?id=952406.

[10] https://bugzilla.mozilla.org/show_bug.cgi?id=1544153.

[11] https://bugzilla.mozilla.org/show_bug.cgi?id=1535880.

[12] https://bugzilla.mozilla.org/show_bug.cgi?id=1544178.

[13] https://bugs.chromium.org/p/chromium/issues/detail?id=942269.

[14] https://github.com/harfbuzz/harfbuzz/issues/2168.

[15] https://bugzilla.mozilla.org/show_bug.cgi?id=1473278.

[16] https://bugs.chromium.org/p/chromium/issues/detail?id=943374.

[17] https://bugzilla.mozilla.org/show_bug.cgi?id=1614250.

[18] https://bugzilla.mozilla.org/show_bug.cgi?id=1615130.

[19] Coverity scan. https://scan.coverity.com/.

[20] Coverity scan: Firefox. https://scan.coverity.com/projects/firefox/.

[21] Google/ClusterFuzz. https://github.com/google/clusterfuzz.

[22] How SQLite is tested. https://www.sqlite.org/testing.html.

[23] KLEE workshop 2018. https://srg.doc.ic.ac.uk/klee18/cfpresentations.html.

[24] Testing Mozilla code. https://developer.mozilla.org/en-US/docs/Mozilla/Testing.

[25] Email correspondence with Ed Maste, Mar. 2019.

[26] Email correspondence with Gordon Tetlow, Apr. 2019.

[27] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible java compiler. In *IWCSCAM*, 2012.

[28] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Sec*, 2009.

[29] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin. Experience report: Developing the Servo web browser engine using Rust. *arXiv:1505.07383*, 2015.

[30] A. Arya, O. Chang, M. Moroz, M. Barbella, J. Metzman, and ClusterFuzz team. Open sourcing ClusterFuzz. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html, 2019.

[31] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE S&P*, 2002.

[32] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *NDSS*, 2011.

[33] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, 2014.

[34] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, 2011.

[35] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *USENIX ATC*, 2019.

[36] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comp. Surv.*, 51(3), 2018.

[37] C. Barrett, A. Stump, C. Tinelli, et al. The SMT-LIB standard: Version 2.0. In *SMT*, 2010.

[38] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *OOPSLA*, 2014.

[39] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world.

CACM, 53(2), 2010.

[40] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in JavaScript bindings. In *IEEE S&P*, 2017.

[41] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *ASPLOS*, 2016.

[42] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries. CMU Tech report CS-07-133, 2007.

[43] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, 2009.

[44] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, 2011.

[45] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[46] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *TISSEC*, 2008.

[47] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *CACM*, 2013.

[48] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, 2015.

[49] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE S&P*, 2012.

[50] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[51] Chrome vulnerability reward program rules. https://www.google.com/about/appsecurity/chrome-rewards/.

[52] The Clang static analyzer. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/clang_static_analyzer.md.

[53] Severity guidelines for security issues. https://chromium.googlesource.com/chromium/src/+/master/docs/security/severity-guidelines.md.

[54] Chromium code coverage. https://chromium-coverage.appspot.com/.

[55] Address sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[56] Memory sanitizer. https://clang.llvm.org/docs/MemorySanitizer.html.

[57] Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[58] H. Corrigan-Gibbs. Personal communication, Feb. 2019.

[59] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS*, 2013.

[60] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling static analyses at Facebook. *CACM*, 2019.

[61] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, 2009.

[62] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[63] D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA*, 2007.

[64] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. 2010.

[65] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.

[66] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *SSPREW*, 2016.

[67] ASan nightly project. https://developer.mozilla.org/en-US/docs/Mozilla/Testing/ASan_Nightly_Project.

[68] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR.

[69] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[70] C. Flanagan, C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[71] M. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole. SMT-based refutation of spurious bug reports in the Clang static analyzer. *arXiv:1810.12041*, 2018.

[72] B. Garmany, M. Stoffel, R. Gawlik, and T. Holz. Static detection of uninitialized stack variables in binary code. In *ESORICS*, 2019.

[73] A. Y. Gerasimov. Directed dynamic symbolic execution for static analysis warnings confirmation. *Programming and Computer Software*, 44(5), 2018.

[74] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[75] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, 2011.

[76] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[77] M. Goregaokar. Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html.

[78] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *ASE*, 2016.

[79] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *FSE*, 2015.

[80] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[81] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *USENIX Sec*, 2013.

[82] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, 1991.

[83] R. Helmer, A. Miyaguchi, and E. Rescorla. Testing privacy-preserving telemetry with prio. https://hacks.mozilla.org/2018/10/testing-privacy-preserving-telemetry-with-prio/, 2018.

[84] D. Hovemeyer and W. Pugh. Finding bugs is easy. *OOPSLA*, 2004.

[85] Joern. https://joern.io/docs/.

[86] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *USENIX ATC*, 2017.

[87] T. Kremenek. Finding software bugs with the Clang Static Analyzer. https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf, 2008.

[88] G. Kwong. JavaScript fuzzing in Mozilla, 2017. https://nth10sd.github.io/js-fuzzing-in-mozilla/.

[89] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Sec*, 2001.

[90] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*, 2018.

[91] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[92] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. *PLDI*, 2017.

[93] K. Li. *Combining Static and Dynamic Analysis for Bug Detection and Program Understanding*. PhD thesis, UMass Amherst, 2016.

[94] The often misunderstood GEP instruction. https://llvm.org/docs/GetElementPtr.html.

[95] https://llvm.org/docs/LangRef.html#phi-instruction.

[96] https://llvm.org/docs/LangRef.html#undefined-values.

[97] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the Linux

kernel using targeted stack spraying. In *NDSS*, 2017.

[98] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[99] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, 2011.

[100] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[101] A. Milburn, H. Bos, and C. Giuffrida. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS*, 2017.

[102] M. Moroz and K. Serebryany. Guided in-process fuzzing of Chrome components. Google Security Blog, 2016.

[103] Mozilla bug bounty program. https://www.mozilla.org/en-US/security/bug-bounty/.

[104] Clang static analysis. https://developer.mozilla.org/en-US/docs/Mozilla/Testing/Clang_static_analysis.

[105] Security severity ratings. https://wiki.mozilla.org/Security_Severity_Ratings.

[106] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[107] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. *JSAT*, 9(1), 2015.

[108] B. S. Pak. *Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution*. PhD thesis, CMU, 2012.

[109] R. Parvez, P. A. Ward, and V. Ganesh. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *CASCON*, 2016.

[110] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *SPIN*, 2004.

[111] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *IEEE S&P*, 2018.

[112] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, 2011.

[113] D. A. Ramos. *Under-constrained symbolic execution: correctness checking for real code*. PhD thesis, Stanford University, 2015.

[114] D. A. Ramos. Personal communication, Aug. 2019.

[115] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Sec*, 2015.

[116] J. Regehr. SQLite with a fine-toothed comb. https://blog.regehr.org/archives/1292.

[117] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *TECS*, 2005.

[118] M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: Testing drivers without devices. In *OSDI*, 2012.

[119] T. Ritter. Adding CodeQL and Clang to our bug bounty program. https://blog.mozilla.org/security/2019/11/14/adding-codeql-and-clang-to-our-bug-bounty-program/.

[120] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM*, 2012.

[121] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at Google. *CACM*, 61(4), 2018.

[122] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, 2015.

[123] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, 15(4), 1997.

[124] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[125] Security at Mozilla. https://wiki.mozilla.org/security.

[126] Semmle. https://semmle.com/.

[127] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESE-FSE*, 2005.

[128] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[129] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX ATC*, 2005.

[130] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[131] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*, 2016.

[132] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.

[133] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. Sok: Sanitizing for security. In *IEEE S&P*, 2019.

[134] SQLite Documentation. The virtual table mechanism of SQLite. https://sqlite.org/vtab.html.

[135] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *PLDI*, 1994.

[136] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[137] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[138] H. Stuart. Hunting bugs with Coccinelle. Master's thesis, University of Copenhagen, 2008.

[139] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE S&P*, 2013.

[140] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *ICSE*, 2018.

[141] E. Van Der Kouwe, V. Nigade, and C. Giuffrida. Dangsan: Scalable use-after-free detection. In *EuroSys*, 2017.

[142] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *OSDI*, 2012.

[143] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *CAV*, 2005.

[144] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, 2005.

[145] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *IEEE S&P*, 2018.

[146] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE*, 2018.

[147] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. In *ICSE*, 2014.

[148] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO*, 2014.

[149] Y. Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[150] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Sec*, 2018.

[151] A. Zaks and J. Rose. How to write a checker in 24 hours. https://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf, 2012.

[152] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl.

[153] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. In *ICSE*, 2015.

[154] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *ACM CCS*, 2017.