SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2019; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Memory Models in Symbolic Execution: Key Ideas and New Thoughts

Luca Borzacchiello, Emilio Coppa*, Daniele Cono D'Elia, Camil Demetrescu

Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Italy

SUMMARY

Symbolic execution is a popular program analysis technique that allows seeking for bugs by reasoning over multiple alternative execution states at once. As the number of states to explore may grow exponentially, a symbolic executor may quickly run out of space. For instance, a memory access to a symbolic address may potentially reference the entire address space, leading to a combinatorial explosion of the possible resulting execution states. To cope with this issue, state-of-the-art executors either concretize symbolic addresses that span memory intervals larger than some threshold or rely on advanced capabilities of modern SMT solvers. Unfortunately, concretization may result in missing interesting execution states, e.g., where a bug arises, while offloading the entire problem to constraint solvers can lead to very large query times.
In this article, we first contribute to systematizing knowledge about memory models for symbolic execution, discussing how four mainstream symbolic executors deal with symbolic addresses. We then introduce MEMSIGHT, a new approach to symbolic memory that reduces the need for concretization: rather than mapping address instances to data as previous approaches do, our technique maps symbolic address expressions to data, maintaining the possible alternative states resulting from the memory referenced by a symbolic address in a compact, implicit form.
Experiments on prominent programs show that MEMSIGHT, which we implemented in both ANGR and KLEE, enables the exploration of states that are unreachable for memory models that perform concretization, and provides a performance level comparable to memory models relying on advanced solver theories.
Copyright © 2019 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Symbolic Execution, Software Testing, Memory Models, Pointer Reasoning

1. INTRODUCTION

Symbolic execution is a technique for program property verification largely employed in the software testing and security domains [1]. By taking on symbolic rather than concrete input values, multiple execution paths can be explored at once, with each path describing the program's behavior for a well-defined class of inputs. Nonetheless, the number of paths to explore can be prohibitively large, e.g., in the presence of unbounded loops. In this article, we tackle one specific problem that may affect exploration in a symbolic executor: *symbolic pointers*.

As in prior works [2], we use the term *symbolic pointer* to refer to any symbolic expression used during the exploration carried by a symbolic engine to reason over the memory state of the program under analysis. In more detail, any expression with at least one non-concrete constituent and that is used by program to perform a memory read or write operation can be regarded under this term. Symbolic pointers may lead an executor to fork the execution, possibly generating an extremely large number of paths. When forks are avoided or at least limited, symbolic executors

*Correspondence to: coppa@diag.uniroma1.it. Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Via Ariosto 25, 00185, Rome, Italy.

Copyright © 2019 John Wiley & Sons, Ltd.

```
1: void bomb(char* a, unsigned char i, unsigned char j) {
2:     char boom;
3:     a[i] = 23;
4:     if (a[j] == 23) boom = 0;
5:     else            boom = 1;
6:     assert(!boom);
7: }
```

Figure 1. Motivating example: can we defuse the bomb?

must resort to either powerful solver theories or pointer concretization. Nonetheless, as we will argue later on in the article (Section 2), these solutions may come with their own drawbacks. Before diving into the discussion of the inner workings of different state-of-the-art solutions, we present a motivating example that should help a reader—likely one knowledgeable regarding symbolic execution techniques, but less familiar with the specific problem tackled in this article—understand the complexity of handling symbolic pointers.

Figure 1 provides the code of function `bomb`. This function takes as inputs an array `a` and two indices `i` and `j`. We assume that `a` can point to a large memory area, possibly the whole memory, and we do not pose any constraint on `i` and `j`. We are interested in characterizing inputs that "defuse the bomb", i.e., do not trigger the `assert` statement. When evaluating line 3, a symbolic executor has to alter the memory state associated with the program in order to track the effects of the write operation `a[i] = 23`. However, unlike a concrete execution where a single byte would be affected by this operation, the pointer `a[i]` can potentially refer to any byte in memory for a symbolic execution. An engine would represent `a[i]` as the sum of two symbolic expressions. While the index `i` is unconstrained and its value can only range between 0 and 255 due to its data type, the array pointer `a` can span an extremely large memory region (in principle, up to 2^{32} bytes in an x86 program and up to 2^{48} bytes in an x86_64 program). In this scenario, tracking the effects of such a write operation is not straightforward. Naively forking the state for each byte possibly touched by this memory access is not feasible in practice due to the massive amount of resources that the symbolic executor would use to keep track of these states. On the other hand, omitting to track the effects of this memory access accurately could impact the ability of the symbolic executor to reason properly later when evaluating the branch condition `a[j] == 23` at line 4. Similar to `a[i]`, pointer `a[j]` can refer to any byte in the whole memory. Although a human can easily understand that only when $i == j$ holds throughout a concrete execution, the `boom` flag would be set to zero and the assertion at line 5 would not trigger, a symbolic executor may fail in practice to generate concrete inputs for indices `i` and `j` that satisfy this condition. Notice also that whenever operations such as the ones at line 4 and line 5 are placed far apart within a program, even a human may have trouble at pinpointing the defusing condition easily.

Contributions. In this article, we first contribute to systematizing key approaches to memory management in symbolic execution by reviewing how four different mainstream implementations of symbolic engines handle memory operations. Our focus is on providing representative examples of orthogonal choices in the design space of memory models. For a more exhaustive overview of symbolic execution tools and their underlying ideas, we refer the reader to a recent survey [1]. To the best of our knowledge, our work provides the first attempt at casting different memory model implementations in a common conceptual framework. Our findings come from the analysis of their source code, and for two of them also from private conversations with their authors; hopefully, they can shed some light on solutions that are deployed in real-world systems, and thus affect thousands of academic researchers and software engineers that use it.

As a second main contribution of the article, our systematization of memory models inspired us to explore a different design perspective in supporting symbolic pointer reasoning: we show how to compactly associate values with *symbolic address expressions* rather than concrete addresses, and investigate how to exploit this idea to implement a fully symbolic memory that can be effectively used in the context of state-of-the-art symbolic executors. A crucial design goal for our approach is that it should be general enough not only to support bug detection, but also to be valuable in application contexts where users are interested in possible consequences of these bugs, e.g., understanding how they can be exploited.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Our approach, which we call MEMSIGHT, natively accounts for merging [3, 4], a mainstream performance enabler in symbolic executors. We integrated MEMSIGHT into the state-of-the-art symbolic executors ANGR [5] and KLEE [6]: an experimental evaluation shows that MEMSIGHT allows for broader state exploration on prominent benchmarks analyzed with ANGR, while it provides valuable runtime speedups when symbolically executing real-world programs under KLEE.

A preliminary version [7] of this work appeared in the Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), where we introduced the main idea behind MEMSIGHT (Section 3) and presented a preliminary experimental evaluation related to MEMSIGHT when integrated into ANGR (Section 5.1.2).

2. MEMORY MODELS

In this section, we address the main practical challenges that a memory model has to face in a symbolic execution engine. After discussing a few core aspects of memory management in symbolic systems, we discuss the peculiarities of the memory models used by four mainstream symbolic execution engines, which we report as representative case studies of different orthogonal approaches. Our goal is to systematize key concepts of previous symbolic memory models, allowing readers to put our work into the perspective of extant approaches. Additional works related to memory models in symbolic execution are instead discussed in Section 6.

2.1. Preliminaries

A symbolic memory model describes the approach used by a symbolic engine to handle memory operations such as *loads* and *stores* performed by the executed program. These could be performed with respect to a specific memory object or, in symbolic engines for low-level code, with respect to a memory address. In this article, we focus on memory models that can deal with programs written in languages where the programmer has unconstrained access to memory, such as C and C++. In this context, memory accesses are often performed through the use of pointers that may potentially refer to any part of the memory, regardless of any constraints that may result from its original source-level data type. As a key issue, memory pointers may become symbolic during the execution of a program: this happens whenever their value depends on the program's input. This dependency could be *direct*, as when an input concurs in the computation of the pointer address (e.g., a symbolic index is used in an array pointer), or *indirect* when the pointer value depends on the choices made during the execution due to the value of one or more inputs (e.g., the address of a dynamically allocated block may depend on the input-dependent sizes of the previously allocated blocks in the execution). Reasoning over symbolic pointers can be especially hard for a symbolic engine since they do not state which memory objects—and which bytes within each memory object—may be impacted by a memory operation. James C. King in his seminal work [8] recognized symbolic pointers as one of the most critical issues in symbolic execution.

Reasoning with Symbolic Pointers. In this section, we discuss the core aspects of a memory model and highlight some of the challenges that contribute at making reasoning on symbolic pointers hard when considering real-world programs.

In general, a memory model should support a symbolic engine in two aspects:

- *memory object reasoning*: it should allow a symbolic engine to load and store values from a *memory object*, i.e., any sequence of bytes that can hold scalar or compound values.
- *memory layout reasoning*: it should allow a symbolic engine to reason over the memory layout, i.e., to identify which memory objects could be involved by a load or a store operation.

To achieve soundness in the case where a load or store accesses a symbolic address, the memory model should be able to capture all possible objects that may be contain the address and, for each object, all possible addresses within the object. Furthermore, the memory model should be able to capture all possible values that could be read from or written to an object.

To better grasp the difference between memory object reasoning and memory layout reasoning, let us consider the three examples of Figure 2. In code snippet (a), a symbolic engine needs to

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

```
1: int data[] = {1, 2};
2: if (data[i] == 2)
3:     target();
```
(a)

```
1: int a = 1, b = 2;
2: if (&a + 1 == &b)
3:     target();
```
(b)

```
1: int a[] = {1}, b = 2;
2: if (a[1] == 2)
3:     target();
```
(c)

Figure 2. Examples of: (a) reasoning on a specific memory object, (b) reasoning on the memory layout, and (c) reasoning on the memory layout and on a memory object. Uninitialized memory is zero-filled.

reason on the content of the array `data` when accessing it using the symbolic index `i`: the function `target` is invoked only if `i` is equal to 1. In this example, the symbolic engine is not interested in the memory location of `data` and `i`, but just in their contents. Nonetheless, even if the memory layout is not crucial, it still needs to correctly map each variable to its corresponding memory object. On the other hand, code snippet (b) shows an example where a symbolic engine needs to explicitly reason on the memory layout (regardless of the content of the pointed memory): only if `b` is placed immediately after `a` in memory, then the function `target` is invoked. Lastly, code snippet (c) combines the two aspects highlighted by the two previous fragments, featuring an example where the symbolic engine has to reason first on the memory layout and then on the content of a memory object: the function `target` is invoked only if `b` – whose value is equal to 2 – is placed immediately after `a` in memory[†].

While reasoning on the memory content is a fundamental requirement of any symbolic engine, reasoning on the memory layout may be less crucial. For instance, a symbolic engine targeting software testing is likely interested in detecting whether an out-of-bounds error occurs, but might not interested in reasoning on the consequences of this error (e.g., understanding which memory object will be accessed when the out-of-bounds memory error occurs). In other words, a symbolic engine for software testing may be interested in detecting bugs, but not concerned with determining how to exploit them. Conversely, memory layout reasoning may be crucial for bug exploitation and vulnerability analysis in software security tools. Another consideration is related to the programming language and the runtime system that the symbolic engine is targeting: there are languages, such as Java, where the raw memory layout may be of a limited relevance for the symbolic engine. For instance, an out-of-bounds error in Java will always throws an exception, regardless of the memory layout chosen by the underlying JVM when placing objects in a real execution. Nonetheless, even when the layout is of limited use, a symbolic engine still needs to identify which memory object is impacted by a memory operation.

2.1.1. Memory Models and Constraint Solvers Constraint solvers are a key component within symbolic execution frameworks. They allow an engine to check whether formulas involving symbolic inputs are satisfiable under a set of path conditions. In order to query a solver, engines have to build formulas in accordance with a specific *logic*. Nowadays, the majority of symbolic executors rely on two well-known logics offered by modern SMT solvers: the theory of bitvectors (`BV`) and the theory of arrays over bitvectors (`ABV`)[‡]. Before discussing how these logics make it possible in practice for an engine to perform memory object reasoning and memory layout reasoning, we briefly review their core aspects.

Theory of Bitvectors (`BV`) The `BV` logic handles closed quantifier-free formulas over the theory of fixed-size bitvectors. Bitvectors allow an engine to model the precise semantics of unsigned and signed two-complement arithmetics. As the name of the theory may suggest, a bitvector has a fixed size—which must be declared at construction time—and can be used to represent constant bits (i.e., a constant bitvector) or symbolic bits (i.e., a symbolic bitvector). The former type is often used to reason over constant data, while the latter for bits coming from an input and whose values will be determined by querying the solver.

[†]For our example to be meaningful, we assume that `b` resides in memory rather than in a CPU register.
[‡]In SMT-LIB[9], the theories of bitvectors and of arrays over bitvectors are denoted as `QF_BV` and `QF_ABV`, respectively. SMT-LIB is an international initiative aimed at facilitating research in Satisfiability Modulo Theories (SMT) that develops and promotes the *SMT-LIB Standard*, a set of common languages that can be implemented across several SMT solvers.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Besides traditional arithmetic and logic operations, BV supports concatenation and slicing of bitvectors, allowing symbolic executors to merge and split them. However, a notable limitation of the slicing operator is that slice boundaries (i.e., upper and lower bit positions) must be concrete offsets, preventing engines from extracting bits through symbolic offsets.

To model the uncertainty on a bitvector value, If-Then-Else (ITE) expressions are offered by BV: the expression ITE(c, v_{true}, v_{false}) evaluates to the bitvector v_{true} when the condition c is satisfied, or to the bitvector v_{false} otherwise. A notable constraint[§] imposed by this logic on ITE expressions is that the v_{true} and v_{false} must have the same size.

Theory of Arrays over Bitvectors (ABV) The ABV logic extends the theory of fixed-size bitvectors by adding reasoning capabilities over arrays [10]. As in most programming languages, an array can be seen as a mapping between indices and values, both implemented as fixed-size bitvectors in this logic. Differently from bitvectors, arrays can contain an unbounded number of elements, thus allowing an engine to not fix its size at construction time. Nonetheless, when defining an array the engine has to declare the bitvector size used for representing indices and values. Hence, the number of elements in an array is implicitly bounded by the maximum value that the index can assume, i.e., $2^n - 1$ where n is the number of bits used for representing the index type.

The ABV logic offers two main operations to deal with arrays. A STORE expression takes as arguments an array expression A, an index IDX, and a value V, and evaluates to a new array expression constructed starting from A by updating the mapping on the index IDX with the value V. On the other hand, a SELECT expression takes as arguments an array expression A and an index IDX, and evaluates to the value V mapped to the index IDX when considering the array A. In both cases, IDX and V have to be bitvectors. To define the initial content of an array, engines can nest STORE expressions that encode the value of each element. Since nesting several STORE expressions can soon make array expressions very long (we discuss possible performance implications in Section 5.2.2), some engines may prefer using *assertions*, i.e., expressions that constrain the initial value of each element. In this case, the array expression is compact but the solver has to consider possibly many additional expressions when reasoning over the array.

2.1.2. Memory Object Reasoning Memory models can naturally build on top of BV or ABV to reason over the content of a memory object. Each memory object could be represented using a bitvector or with an array (which in turn uses bitvectors to represent array elements).

The BV logic is well-suited for operating on bit- or byte-sized objects but it does not support reasoning over symbolic indices when extracting bits from bitvectors. To cope with this limitation, ITE expressions can be used by symbolic executors to conditionally describe the content of an object. For instance, consider a memory object B made of two bytes (i.e., a 16-bits bitvector). When dealing with a symbolic pointer IDX that could potentially refer any of the two bytes in this object, a memory model could resort to the expression `ITE(IDX == 0, B[7:0], B[15:8])` meaning that the byte resulting from the memory operation is equal to the first byte of the bitvector B if the symbolic pointer IDX is equal to 0, or equal to the second byte of B otherwise. Notation `[X:Y]` on a bitvector represents the slicing operation, extracting the bits from position Y to X. Since BV only deals with constant numeric values and with fixed-sized bitvectors, IDX must adhere to one of these two types. ITE expressions built following this approach have to explicitly enumerate the values that could result from their evaluation. This may make it easier for an engine to implement and apply rewriting rules able to compact and optimize expressions.

Since ITE expressions can soon become very lengthy in presence of pointers ranging over large objects, several symbolic executors choose to resort to ABV instead of BV, taking advantage of the support for array expressions. By nesting SELECT and STORE expressions, a symbolic execution engine is able to describe the content of a memory object in a natural way across sequences of memory accesses even when dealing with symbolic indices, without the need of building ITE expressions. For instance, when dealing with a symbolic pointer IDX that could potentially refer to

[§]For instance, ITE expressions returning objects of different size could be used to support symbolic object sizes.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

any byte within an object `A`, a memory model could construct the expression `SELECT(A, IDX)` where `A` is an array expression and `IDX` a bitvector object. The array `A` could be an empty mapping (i.e., `A = ARRAY()`), or an array expression with one or more nested `STORE` expressions (e.g., `A = STORE(STORE(ARRAY(), 0, 10), 1, 20)`).

Most symbolic executors that build on top of the `ABV` logic define memory objects using arrays that map indices chosen for simplicity as 32-bit long to 8-bit values. As a consequence, these arrays can potentially store an extremely large number of elements. This contrasts with engines relying only on the `BV` logic, where bitvectors are often sized according to the actual size of the objects that they represent. Nonetheless, an engine can pose a limit on the maximum number of elements stored in an array by adding constraints at query time on the highest index value. These constraints can be changed across different queries, providing interesting opportunities for symbolic explorations involving objects with a symbolic size.

2.1.3. Memory Layout Reasoning A fundamental component in a symbolic execution engine is the *symbolic store* [4]. This is often described as a mapping between program variables and their values that allows an engine to identify which objects are impacted by a memory operation, supporting memory layout reasoning. However, this definition is an oversimplification that does not fit the complexity of real-world programs.

Modern applications are typically structured as functions and object classes. To correctly tell apart different values of local variables and parameters for different pending invocations of the same function (think, e.g., of recursive calls), a symbolic store should incorporate the notion of activation frame, allowing the engine to access the variable instance within the current context.

When considering low-level code, the situation becomes even worse: variables are often referred to by a piece of code using their memory addresses, but the same locations could host different variables throughout the execution of a program (think, e.g., of the stack). Additionally, binary code may be generated by modern compilers in order to access only bytes of an object that are relevant for the computation (for instance a data structure field). In other words, any information about the data type, such as the overall size of the accessed object, is often lost during compilation. In practice, it can be very hard to logically split the memory layout into different memory objects when analyzing binary code since the exact boundaries between objects could be unknown at run time.

For these reasons, very different approaches for performing memory layout reasoning have been devised by different symbolic executors. We can classify these approaches with respect to how they treat the memory: as we will discuss in Section 2.2, some executors see the memory as a *single flat memory area*, while others split it into *different regions*.

To track memory objects in the memory layout, regardless of how memory is partitioned, engines may explicitly maintain a mapping between addresses and objects, or delegate this task entirely to the solver. Maintaining full control over the symbolic store may allow an engine to identify objects impacted by a memory operations without resorting to the solver. However, this may add significant implementation complexity, making it hard to deal with symbolic pointers. In fact, a mapping of addresses may not be sufficient to discriminate which regions or objects are affected by a symbolic access, requiring an invocation of the solver. For this reason, an engine may resort to ITE expressions to deal with uncertainty (see., e.g., Section 2.2.1) or rely on the capabilities of the `ABV` logic (see, e.g., Section 2.2.3), where symbolic pointers can be modeled as symbolic indices. However, even in the latter case, to avoid querying a solver for each memory operation, executors often implement internal data structures to avoid interactions with the solver when no uncertainty arises on pointers, possibly reducing the runtime overhead. Hence, the picture we get in practice is blurred: no executor relies merely on its internal mapping to perform memory layout reasoning, and no executor delegates this task entirely to the solver.

2.2. Case Studies

While symbolic execution is one of the most powerful techniques for analyzing the behavior of a program, its memory management is fraught with significant practical implementation issues. In this section, we discuss the memory models of four mainstream symbolic execution engines

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

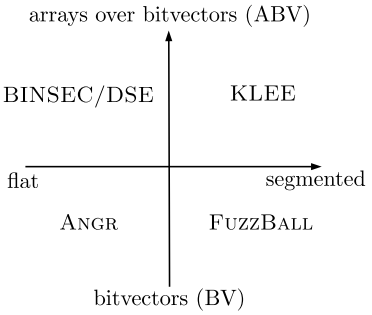

Figure 3. Classification of the memory models implemented in ANGR, KLEE, FUZZBALL, and BINSEC with respect to the approach adopted for memory layout reasoning (x-axis) and memory object reasoning (y-axis). The x-axis depicts how a model devises the memory layout: *flat* means that the memory is treated a single flat region, while *segmented* means that the model is organized as a set of distinct objects. The y-axis depicts which logic (BV or ABV) is used to perform memory object reasoning.

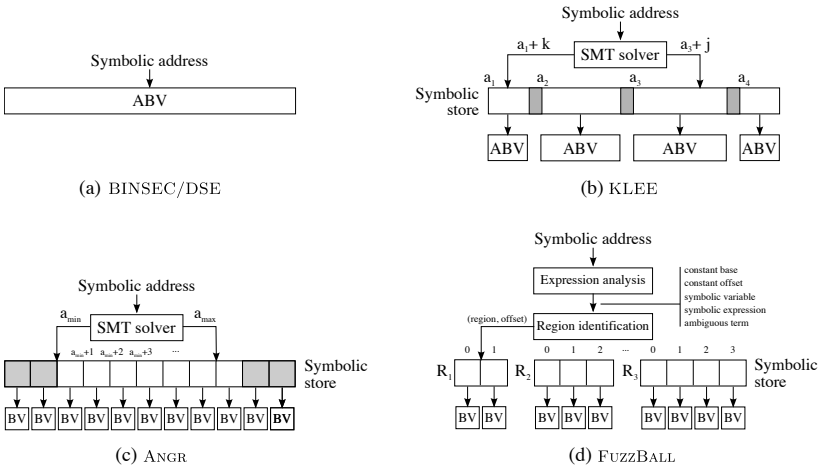

Figure 4. Memory abstractions for: BINSEC, KLEE, ANGR, and FUZZBALL.

that target programs either available in a binary format or translated into a compiler's intermediate representation. In particular, we consider ANGR [5], KLEE [6], BINSEC [11], and FUZZBALL [12]. These frameworks provide an interesting overview about different choices that could be made in the design of a memory model in a symbolic engine able to deal with low-level code where symbolic pointers can possibly refer to any part of the program memory. Throughout our discussion, we classify them according to two aspects (see Figure 3). From one side, we consider the logic needed by these models to support memory object reasoning. Some of them can work using the BV logic, while others has strong requirements and rely on the ABV logic. On the other side, we also consider how they support memory layout reasoning, e.g., whether memory is split into several segments or conceived as a single flat memory region. The memory abstractions provided by the four considered frameworks are depicted in Figure 4.

2.2.1. ANGR. ANGR [5] is an open-source framework for binary analysis. It lifts binary code into the VEX intermediate language [13], thus providing an architecture-independent symbolic execution engine. Although it can offload code execution to the Unicorn CPU emulator and it can also be used to analyze instruction traces, ANGR can be mainly regarded as a static symbolic executor based on the Z3 [14] constraint solver. During the DARPA Cyber Grand Challenge, ANGR has been used as a key component within the Mechanical Phish Cyber Reasoning System [15], proving its strong capabilities to reverse engineering and security practitioners.

ANGR represents memory objects through the use of bitvectors as shown in Figure 4 (c). The symbolic store is maintained internally as a mapping between concrete addresses and bitvector

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

expressions. Whenever the memory content at an address is not constant, but depends on the value of one or more symbolic inputs, ITE expressions are used to conditionally describe the byte content. Since binary code may fail to provide any kind of information about the object sizes, ANGR takes the extreme approach where each byte of memory can be seen as a distinct memory object, regardless of the data boundaries that may have existed in the original source code. Nonetheless, memory operations involving multiple subsequent bytes (e.g., a 4-byte load) are handled by ANGR as a single operation for performance reasons.

For these reasons, Figure 3 places ANGR in the left bottom corner, highlighting a memory object reasoning based on the bitvector logic (BV) and a memory layout reasoning based on a flat model.

To make the symbolic execution scale to real-world programs, ANGR trades performance for soundness by implementing a *partial* memory model that closely resembles the approach originally proposed in MAYHEM [16]. When a symbolic pointer is used within a memory operation, ANGR queries the Z3 SMT solver for computing the maximum and minimum values that the address can assume under the current path conditions. These values are used to evaluate how large the memory chunk spanned by a symbolic address is. If the memory chunk is larger than one byte (i.e., the address is symbolic and can assume multiple values) and the pointer is used within a write operation, then the address is concretized to the maximum value it can assume[¶]. In other words, symbolic pointers used in write operations are always concretized in ANGR. On the contrary, symbolic pointers used in read operations get a different treatment. If the memory chunk spanned by a read operation is larger than a fixed threshold[†], the symbolic address is concretized considering any valid solution returned by the SMT solver. Otherwise, ANGR asks Z3 to enumerate all the solutions (i.e., addresses possibly touched by the read operation), constructing an ITE expression that captures the uncertainty. Notice that, as a result of this treatment, ANGR generates ITE expressions that have a bounded number of cases when handling a read operation, i.e., the number of cases considered by the ITE cannot exceed the concretization threshold.

Address concretization makes evaluation for a symbolic engine easier, since possibly complex expressions are replaced with combinations of concrete values, often reducing the solving time for queries submitted to the SMT solver. However, this strategy sacrifices completeness for scalability. When a symbolic pointer is concretized, the symbolic engine is possibly discarding many interesting program states. For this reason, ANGR optionally allows a user to customize the threshold used for read concretizations and also to treat write operations similarly to read operations. In this case, a symbolic write address is concretized only when the spanned memory chunk is larger than another user-defined threshold. Otherwise, ANGR asks Z3 to enumerate all the solutions and, for each valid address, updates the mapping inside the symbolic store: a new ITE expression is constructed, which conditionally evaluates to the new value if the symbolic address equals the address, or the previously stored value otherwise. Although ANGR can be tuned in order to accurately deal with a write symbolic address, it will likely not scale when the spanned memory chunk is very large. ANGR needs to perform two expensive operations: enumerate all valid solutions through the SMT solver and sequentially update their mapping inside the symbolic store.

A significant benefit of this memory model is that merging two memory stores is a straightforward operation. Merging promotes scalability by allowing engines to minimize the memory overhead needed for keeping track of *similar* paths. This also allows avoiding repetitive analysis work, i.e., different paths exploring the same piece of code can be analyzed as a single path after a merge operation. To identify the paths to merge, ANGR integrates the *veritesting* merging strategy [4].

From the point of view of the memory model, it is interesting to understand how ANGR operates when the symbolic stores of two states need to be merged. It starts by cloning one of the two states, electing the copy as the merged state. To accurately update its symbolic store, ANGR scans the entire address space, checking for each byte if the memory mappings in the two states differ. When

[¶] ANGR has been often used for finding inputs able to crash an application: concretizing an address to its maximum value may maximize the chances of this event.

[†] ANGR uses a threshold equal to 1024 bytes. This value was originally proposed by MAYHEM as a reasonable trade-off between scalability and accuracy.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

this condition is detected, the mapping in the merged state is updated with an ITE expression that conditionally describes the memory content depending on the path constraints of two states. For instance, given the states s_1 and s_2 to merge, the content at an address a in the merged state can be updated with the expression ITE($s_1.\pi$, v_1, v_2) where $s_1.\pi$ is the set of path constraints in s_1, v_1 is the byte content at address a in s_1, while v_2 is the byte content at address a in s_2. To make the merge operations efficient, ANGR developers have carefully designed the symbolic store in order to make it easy to detect when two states have one or more bytes with a different content.

2.2.2. KLEE. KLEE [6] is an open-source symbolic virtual machine built on top of the LLVM compiler infrastructure. It cannot evaluate directly binary code as done by ANGR, but evaluates programs translated into the LLVM Intermediate Representation (IR): this makes it possible for KLEE to potentially analyze code written for different architectures and different programming languages. To reason over symbolic expressions, KLEE may rely on several popular SMT solvers, such as STP [17] and Z3 [14]. Additionally, it can translate its queries into the SMT-LIB format, supporting any solver compliant with this language. KLEE has been used in many academic and industrial projects [18, 19], demonstrating its reliability and flexibility in many application contexts.

KLEE represents memory objects using `ABV` instances as illustrated in Figure 4 (b). The symbolic store is implemented as a mapping between concrete addresses and `ABV` expressions. However, differently from ANGR, which tracks in its mapping the content of each byte in memory, the symbolic store in KLEE only takes into account base address and size for each memory object. Since the LLVM IR representation of a program does not dictate the process memory layout[‖], KLEE uses a memory allocator to choose the base address where a memory object should be stored, concretizing the object size when symbolic. However, since this allocator does not take into account the memory region where the objects will be hosted in a concrete execution (e.g., stack or heap region), KLEE reasons on a concrete layout that may not be representative of a possible layout in a concrete execution of the program, possibly leading to false positives (uncompleteness). KLEE uses this artificial layout merely for reasoning over object boundaries that for sake of simplicity are mapped into a linear address space. We note that the memory layout concretization of KLEE may introduce false negatives as well (unsoundness), e.g., if a bug is not revealed under the chosen concrete layout. For these reasons, Figure 3 places KLEE in the right upper corner, highlighting a memory object reasoning based on the array logic (`ABV`) and a memory layout reasoning that treats the memory as a set of independent memory objects.

When a load or store operation involves a symbolic address, KLEE resorts to the SMT solver to determine which memory objects may be impacted by the memory access. To this end, KLEE queries it to get a valid assignment for the symbolic expression representing the pointer, evaluating where the obtained concrete address falls within the memory layout. Given the concrete address, KLEE determines which memory object o is affected by the memory access. It then checks whether another object could be impacted under a different assignment of the input: KLEE queries the solver looking for another assignment that makes the pointer fall outside the boundaries of the memory object o. This process is repeated as long as the solver is able to provide assignments that fall outside previously considered objects. A state is then forked for each memory object that is proven to be affected by the symbolic pointer, constraining the pointer expression to the boundaries of the memory object for that state. The theory of arrays is then used to reason over the effects of the memory access within a memory object for a given state.

Besides considering valid accesses to active objects, KLEE checks also whether the pointer could fall outside any active memory object, i.e., it checks whether an invalid memory access is possible. Since a symbolic pointer may potentially refer any object in the memory layout, in the worst case scenario KLEE forks $n + 1$ states, where n is the number of memory objects that are active at the time of the memory access, while the additional state is used to model the invalid access scenario.

[‖]For instance, the LLVM IR does not impose a relative position among memory objects allocated on the stack through the `alloca` instruction, leaving this decision to the runtime system. On the other hand, binary code typically refers to stack variables through relative offsets that are computed with respect to the stack pointer or the stack base pointer.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

A notable downside of this memory model is that merging the memory stores of two states can be very complex. KLEE is able to correctly merge two memory stores only when they have exactly the same memory layout, i.e., the symbolic stores are the same and only the object contents may differ. KLEE follows a merging strategy similar to the one implemented in ANGR but at an object granularity rather than in the full address space. In other words, if the number of memory objects differs between the two states, or the size of a common memory object differs among the two states, then KLEE cannot merge them since the two memory layouts cannot be easily reconciled.

2.2.3. BINSEC. BINSEC [11] is an open-source platform for formal analysis of binary codes. It provides a *dynamic symbolic execution* (DSE) engine that evaluates traces recorded using dynamic binary instrumentation [20]. Traced instructions are lifted to a custom intermediate representation, called DBA, allowing BINSEC to support different architectures. Several solvers can be used with this framework thanks to the support of the SMT-LIB query format. While BINSEC implements different memory models [21], we present details of the one used by default in the DSE engine since it adds an interesting perspective in our discussion.

As shown in Figure 4 (a), BINSEC/DSE treats the whole memory as a single `ABV` instance using the theory of arrays to handle load and store operations. Each state keeps track of a single formula that describes the content of the memory: the formula is a `STORE` expression that *recursively* keeps track of the sequence of store operations that have been performed during the execution, starting from the initial memory state of the program under analysis. To scale over long program executions, BINSEC/DSE applies several optimizations and rewrite rules [22], such as *read-over-write*, *write-over-write*, *constant propagation*, and *memory flattening*. For instance, the read-over-write rule is used to allow BINSEC to determine the result of a `SELECT` operation whenever only concrete indices are present in the expression, preventing it from submitting possibly expensive queries to the solver. Despite these optimizations, queries generated by this memory model can be very complex, imposing a heavy burden on the solver. However, since a DSE engine should ideally submit just a few queries during the evaluation of an execution trace, this model may provide a reasonable trade-off between performance and analysis accuracy. When instead considering this model within the context of a static symbolic execution engine where multiple states are maintained in parallel, it is not straightforward to understand how to efficiently perform state merging since the memory content is described using a single formula, making it difficult for engines to pinpoint memory differences in the two states.

Since BINSEC/DSE treats memory as single flat region and the theory of arrays is used to reason over symbolic accesses, its memory model appears in the left upper corner of Figure 3.

2.2.4. FUZZBALL. FUZZBALL is an open-source framework for symbolic execution of binary code. Its name derives from the phrase *FUZZing Binaries with A Little Language*, where *little language* refers to the *Vine* intermediate language built on top of the VEX IR and used by FUZZBALL to reason directly on binary code. The framework can directly interface with the STP solver but, similarly to other engines, other solvers are supported via the SMT-LIB query format.

FUZZBALL splits the memory address space into different regions. Initially, a single region is available and it is used to handle accesses over concrete addresses. Other regions are instead dynamically created by FUZZBALL in order to deal with symbolic accesses. In particular, when a memory access is performed, FUZZBALL analyzes the expression representing the memory address syntactically, decomposing it into different terms, whose type can be *constant base*, *constant offset*, *symbolic variables*, *symbolic expressions*, or *ambiguous terms*. For instance, a constant term within an expression is classified as a constant offset when lower than `0x4000`, or as a constant base otherwise. Since this classification is done through pattern matching rules that are designed to correctly classify expressions that are more likely to occur in a program, it may fail when analyzing complex expressions, marking unexpected subexpressions as ambiguous. Depending on the result of this classification, FUZZBALL checks whether the base address of the pointer expression is constant and thus it can be handled by reasoning on the initial region, or the base address is symbolic or ambiguous and FUZZBALL cannot understand which concrete memory address is possibly referred

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

by this pointer. In this case, FUZZBALL checks if there exists a previously created region that is associated to a symbolic subexpression contained within the current pointer expression. If this happens, FUZZBALL handles the memory access by reasoning on that region, otherwise a fresh new region is created. Note that any region that is dynamically created by FUZZBALL when evaluating a program is not mapped to a specific concrete address. Instead, FUZZBALL associates these regions to specific symbolic expressions. This allows it to reason on memory operations that consistently involve the same symbolic expression. For instance, a loop referring the symbolic expression $\alpha + i$, where α is symbolic and i is constant but changes at each iteration, would be handled reasoning always on the same region.

When the region impacted by a memory access is identified, FUZZBALL resorts to an interesting strategy, known as *table treatment*, to deal with symbolic accesses. Each region can be seen as as mapping between concrete offsets and either concrete or symbolic values. When a symbolic store is performed by a program, FUZZBALL behaves similarly to ANGR: for each constant offset within the region that may be referred by the store operation, FUZZBALL updates the value associated to the offset with an ITE expression that conditionally returns the new value if the symbolic address matches the offset in the region, or the old value otherwise. When a load operation is performed, FUZZBALL enumerates the offsets potentially referred by the symbolic address and recovers their values from the region. This mapping between offsets and values is later used to generate an expression that can be handled by a solver. If FUZZBALL is configured to use the theory of arrays, then a new array is defined, setting its initial content based on the constructed mapping, and a `SELECT` expression involving the array and the symbolic address is returned. Otherwise, relying only on the `BV` logic, FUZZBALL constructs an ITE expression that returns the correct memory content depending on the value of the offset. However, differently from ANGR, which constructs expressions resulting in abstract syntax trees (AST) that are degenerate, FUZZBALL constructs a balanced AST by branching the ITE expression for each bit of the symbolic address. Similarly to the partial memory model adopted in ANGR, FUZZBALL performs table treatment only when the memory area spanned by a symbolic pointer is less than a constant threshold. Otherwise, FUZZBALL concretizes the symbolic address using a heuristic.

We have placed FUZZBALL in the lower right corner of Figure 3 since it views memory as a collection of regions and exploits by default the theory of bitvectors to reason over memory objects. Although the concept of memory regions offers an elegant solution for dealing with symbolic pointers, this approach nonetheless does not provide a general solution to this problem. While FUZZBALL's memory model is not amenable to state/path merging, it does not perform concretizations that may impact soundness.

2.2.5. Wrap-up. In the previous sections we have presented the memory models of four mainstream symbolic execution engines. These approaches provide quite different solutions to the problem of dealing with symbolic pointers. The theory of arrays natively supports reasoning over symbolic indices, but makes it really hard for an engine to support state merging. Representing the whole memory using a single array as in BINSEC/DSE can lead an executor to generate very complex queries, preventing engines from adopting this approach in the context of static symbolic execution. KLEE models each memory object using a distinct array, but then it has to fork the execution into several states when a symbolic access may refer to different objects. Additionally, KLEE considers a fixed memory layout that does not fit well when exploitation is the ultimate goal of symbolic exploration. ANGR implements a partial memory model that can miss interesting states due to premature concretization. While FUZZBALL provides an elegant solution for dealing with fully symbolic accesses, reasoning simultaneously over different regions is not supported. State/path merging is implemented in ANGR and, to some extent, in KLEE, but neither in FUZZBALL nor in BINSEC/DSE. These different design choices are due to the fact that, although merging reduces the number of paths/states, it may lead to more complex formulas that could slow down SMT queries.

In the next section, we present a novel memory model that does not build on top of the theory of arrays, but can still deal with write operations involving symbolic pointers. A requirement for our model is to support state merging as a scalability tool as discussed in [4]. Additionally, our model

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

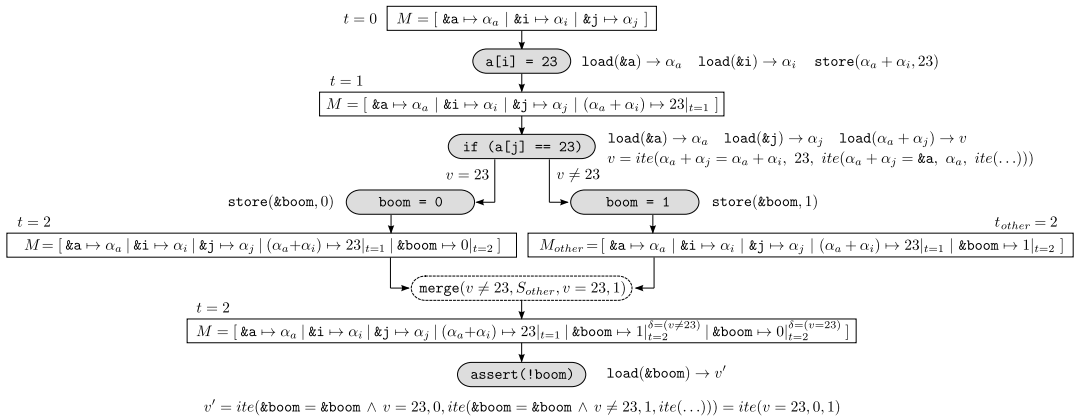

Figure 5. Symbolic execution of the bomb example of Figure 1. Expressions `&a`, `&i`, and `&j` denote the concrete addresses of the corresponding variables. The bomb does not detonate only if the `assert` succeeds, i.e., if $v' = 0$, that is $v = 23$, which happens for instance if $\alpha_i = \alpha_j$.

should be general enough to support accurate reasoning on the memory layout, allowing users to use it even in the context of software vulnerability detection and exploitation.

3. TECHNIQUE

To illustrate how MEMSIGHT works, we start from a simple base version and then we refine it to make it more general and efficient in practice. We target a general setting in which a symbolic engine maintains for each explored state a set of *path constraints* π reflecting path choices taken at each branch based on the values of symbolic inputs. An SMT solver is invoked to check for path feasibility at branch instructions and to retrieve models for symbolic values and expressions, e.g., to apply concretization. In our formalization, data can be stored in memory through *load* and *store* operations over expressions describing addresses. Similarly to ANGR and BINSEC/DSE, our approach treats the memory as a single flat memory region, although it may also be used to represent individual objects as in KLEE (Section 4.2). Unless otherwise stated, we assume that all addresses and values are expressions over concrete and/or symbolic terms. Furthermore, an engine may decide to *merge* the effects from multiple paths into one to seek for efficiency, thus requiring that the respective memories be merged as well.

3.1. Base Version

We model our symbolic memory M as a set of tuples (e, v, t, δ), where e is an expression that denotes an address and v is an expression that denotes the value at address e. Attribute t is the logical time at which the tuple was created and is used by load operations to determine the latest value written at a given address. To support a merging operation for memories, we account for a predicate δ that captures the conditions under which the tuple is valid: δ is typically computed by the executor in terms of diverging path constraints between the states that are to be merged.

The base version of our memory model is shown in Algorithm 1. To explain how it works, consider again the example of Figure 1. In order to determine whether there is any bomb-defusing input, we set up a symbolic executor to associate pointer `a` with symbolic value α_a and indices `i` and `j` with symbolic values α_i and α_j, respectively.

The program's effects on the symbolic state are illustrated in Figure 5. To keep track of logical time, the state includes a timer t that starts at zero. Initially, the memory M includes the address-value mappings resulting from parameter passing. For the sake of compactness, we denote tuple (e, v, t, δ) as $e \mapsto v|_{t=\ldots}^{\delta=\cdots}$, omitting t if $t = 0$, and δ if $\delta = true$.

Memory Loading and Storing. To perform the assignment `a[i] = 23`, the program first loads the values of variables `a` and `i`. A `load`(e) operation (Algorithm 1) builds an ITE expression that

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Algorithm 1 MEMSIGHT – base version

M := symbolic memory (initially empty)

t := timer (initially 0)

```
1: function STORE(e, v):
2:     t ← t + 1
3:     M ← M ∪ {(e, v, t, true)}
4: end function

1: function LOAD(e):
2:     v ← 0
3:     for x ∈ M by ascending timestamp do
4:         v ← ite(e = x.e ∧ x.δ, x.v, v)
5:     end for
6:     return v
7: end function

1: function MERGE(δ, S_other, δ_other, t_a):
2:     for {x ∈ M | x.t > t_a} do
3:         M ← M|_{x.δ ← x.δ ∧ δ}
4:     end for
5:     for {x ∈ S_other.M | x.t > t_a} do
6:         x.δ ← x.δ ∧ δ_other
7:         M ← M ∪ {x}
8:     end for
9:     t ← max(t, S_other.t)
10: end function
```

attempts to match e against all addresses previously assigned in M, considering the most recent tuples first. The "else" case of the innermost ITE accounts for uninitialized memory locations, which for the sake of simplicity we consider set to zero by default in this base version. In our example, `load(&a)` yields $ite(\texttt{\&a} = \texttt{\&a}, \alpha_a, ite(\texttt{\&a} = \texttt{\&i}, \alpha_i, ite(\texttt{\&a} = \texttt{\&j}, \alpha_j, 0)))$, which can be simplified to α_a. Similarly, `load(&i)` yields α_i. The assignment is done by a `store`$(\alpha_a + \alpha_i, 23)$ operation that adds $(\alpha_a + \alpha_i, 23, 1, true)$, i.e., $\alpha_a + \alpha_i \mapsto 23|_{t=1}$, to M after updating t.

The test `if (a[j]==23)` performs a `load`$(\alpha_a + \alpha_j)$ operation, which constructs an ITE expression v that selects the appropriate value at address $\alpha_a + \alpha_j$. This is done by first matching $\alpha_a + \alpha_j$ against the most recent written symbolic address, i.e., $\alpha_a + \alpha_i$, and later considering parameters `&a`, `&i`, and `&j`. The execution then forks a new state $S_{other} = \{M_{other}, t_{other}\}$ for the "else" branch, where M_{other} is a clone of M and $t_{other} = t$. This branch is taken iff $v \neq 23$.

State Merging. As the value of the `boom` variable depends on the taken branch, a `merge` operation (Algorithm 1) reconciles the states by fusing M and M_{other} into M. The operation takes four parameters: S_{other} and $\delta_{other} = (v \neq 23)$ are respectively the state and path condition resulting from the "else" branch, while $\delta = (v = 23)$ is the path condition of the "then" branch that kept on working on M ; finally, $t_a = 1$ is the timestamp of the immediate dominator of the two branches.

The merge updates all tuples added to M since the branch point at time t_a by guarding them with the "then" branch condition δ (lines 2–3), and then adds to M all tuples added to M_{other} since t_a (i.e., their timestamp is higher than t_a), guarding them with the "else" branch condition δ_{other} (lines 4–6). In our example, this results in having tuples $\texttt{\&boom} \mapsto 1|_{t=2}^{\delta=(v\neq 23)}$ and $\texttt{\&boom} \mapsto 0|_{t=2}^{\delta=(v=23)}$ present in M after merging the states (Figure 5).

Finally, the program loads and returns the value of `boom`, building the (simplified) expression $v' = ite(v = 23, 0, 1)$. Symbolic execution can therefore conclude that any model of $v' = 0$, e.g., such that $\alpha_i = \alpha_j$, will defuse the bomb.

3.2. Refinements

In its initial naive formulation, the proposed scheme suffers from a few generality and performance issues. We now present a number of refinements that are crucial to make MEMSIGHT effective in practice. Algorithm 2 integrates some of these refinements, helping the reader grasp these ideas when immersed into MEMSIGHT.

Address Range Selection. One of the main drawbacks of Algorithm 1 is that `load` and `merge` operations need to scan the entire memory, which can be highly time and space-consuming. Observe that it is common for a symbolic address to be constrained within a certain interval [16]. Hence, a more effective approach is to index each tuple (e, v, t, δ) with the smallest range $[a, b]$ that includes all possible values e can attain (line 6 of `store` in Algorithm 2). The range can be computed by the SMT solver (lines 2–3). A `load`(e) operation can therefore scan only those tuples whose ranges intersect with the interval spanned by the minimum and maximum values of e (lines 2–3, 8).

One possible way to support `range` operations efficiently is to maintain an *interval tree* [23] to allow for efficient retrieval of all stored intervals that overlap with a given one. However, we should

Copyright © 2019 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

keep in mind that when a branch is encountered, an executor typically clones the state along with the associated memory. To optimize space usage, we thus propose a memory-wise *paged interval tree*. We partition the address space in pages: a primary interval tree built on top of page indices holds pointers to secondary interval trees that contain the tuples in M. Each tuple is contained in exactly one secondary tree. The page size is empirically determined to minimize the maximum tree size in the data structure. Both the primary and the secondary trees are maintained using a copy-on-write strategy that minimizes the need for cloning and promotes memory sharing among different states.

Multi-Byte Load and Store. The solutions presented in this section are designed for 1-byte memory objects. For the sake of simplicity, in our discussion we even implicitly treated α_a as 1-byte object, which is unrealistic since it represents a memory address. Multi-byte operations can be supported by issuing separate `store` and `load` operations for individual bytes and combining the results. For instance, a $\texttt{load}(e, \texttt{sizeof(int)})$ can be obtained by concatenating the values produced by $\texttt{load}(e)$, $\texttt{load}(e+1)$, $\texttt{load}(e+2)$, and $\texttt{load}(e+3)$. This strategy is adopted by several symbolic executors such as KLEE.

Concrete Memory. One crucial aspect to take into account is that the majority of memory accesses in a symbolic exploration typically happen on *concrete addresses*. Capturing concrete stores in an interval tree would result in maintaining information about many ranges of size 1. We thus extend our representation with a concrete memory object that associates concrete addresses with expressions representing values. Each expression is annotated with a timestamp, so it can possibly be combined with values mapped to symbolic addresses during a load operation. For the sake of efficiency, concrete memory can be implemented as a paged hashmap with copy-on-write cloning for pages, similarly to ANGR's default memory implementation.

Memory Clean-up. Algorithm 1 naively creates one tuple for every `store` operation. A useful improvement is getting rid of older tuples that are no longer needed: one approach is to remove a tuple if its address is "equivalent" to the one being written (line 5 of `store` in Algorithm 2), i.e., they lead to the same concrete address for any possible valuation of symbols.

Symbolic Uninitialized Memory. Identifying how a program may behave when accessing uninitialized memory regions is crucial for testing and vulnerability exploitation. In our base version, we have assumed that an uninitialized cell holds a zero value, which limits the precision of the analysis. The $\texttt{load}(e, v)$ operation of Algorithm 2 supports symbolic uninitialized memory by performing an *implicit store* that assigns a new symbol to address e if e is not fully "covered" by address expressions already in M (lines 4–6). A subtle issue is how to make sure that accessing an uninitialized memory address consistently yields the same symbolic value over time. More precisely, for any two $\texttt{load}(e) = v$ and $\texttt{load}(e') = v'$ operations, if γ is a valuation of symbols such that $\gamma \models e = e' = x$ and address x is uninitialized, then $\gamma \models v = v'$. To achieve this property, we use a tie-breaking strategy based on negative timestamps for tuples created by implicit stores. Observe that our treatment of uninitialized memory shares similarities with how constraint solvers deal with uninterpreted functions.

Compacting Load Expressions. Broad pointers, i.e., symbolic pointers that may range over a large interval of the address space, often span memory regions containing zero-initialized values or another user-defined value (e.g., when setting bytes to a constant value such as -1 to mark invalid data). MEMSIGHT exploits this practical observation in the `load` primitive by compacting ITE expressions that enumerate values in such memory regions. In particular, MEMSIGHT merges ITE cases mapped to the same value when they refer to consecutive concrete memory addresses. For instance, in presence of an ITE expression such as $ite(e = \texttt{0x0}, v, ite(e = \texttt{0x1}, v, ite(e = \texttt{0x2}, v, ite(e = \texttt{0x3}, v, v'))))$, MEMSIGHT generates an expression of the form $ite(\texttt{0x0} \leq e \leq \texttt{0x3}, v, v')$. For this optimization to be sound, MEMSIGHT takes into account timestamps and merging conditions, excluding cases that cannot be merged. To apply this optimization more effectively, tuples collected from the concrete memory are sorted first by timestamp and then by addresses. MEMSIGHT is able to generate compact ITE expressions even when multiple regions with different values are affected by a load operation. For instance, it can optimize an expression such as $ite(e = \texttt{0x0}, v, ite(e = \texttt{0x1}, v, ite(e = \texttt{0x2}, v', ite(e = \texttt{0x3}, v', v''))))$ into $ite(\texttt{0x0} \leq e \leq \texttt{0x1}, v, ite(\texttt{0x2} \leq e \leq \texttt{0x3}, v', v''))$.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Algorithm 2 MEMSIGHT – improved version

M := symbolic memory (initially empty)
π := current path constraints in symbolic engine
t := positive timer (initially 0)
$\bar{t}$:= negative timer (initially 0)
$sat(\psi) \triangleq \exists\gamma : \gamma \models \psi$
$equiv(e, e', \pi) \triangleq unsat(e \neq e' \ \wedge \ \pi)$
$range(M, a, b) \triangleq \{x \in M | [x.a, x.b] \cap [a, b] \neq \emptyset\}$

1: **function** MERGE($\delta, S_{other}, \delta_{other}, t_a, \bar{t}_a$):
2: **for** $\{x \in M \mid x.t > t_a \vee x.t < \bar{t}_a\}$ **do**
3: $x.\delta \leftarrow x.\delta \wedge \delta$
4: **end for**
5: **for** $\{x \in S_{other}.M \mid x.t > t_a \vee x.t < \bar{t}_a\}$ **do**
6: $x.\delta \leftarrow x.\delta \wedge \delta_{other}$
7: $M \leftarrow M \cup \{x\}$
8: **end for**
9: $t \leftarrow max(t, S_{other}.t)$
10: $\bar{t} \leftarrow min(\bar{t}, S_{other}.\bar{t})$
11: **end function**

1: **function** STORE(e, v):
2: $a \leftarrow min(e, \pi)$
3: $b \leftarrow max(e, \pi)$
4: $t \leftarrow t + 1$
5: $M \leftarrow M \setminus \{x \in range(M, a, b) \mid equiv(e, x.e, \pi)\}$
6: $M \leftarrow M \cup \{(a, b, e, v, t, true)\}$
7: **end function**

1: **function** LOAD(e):
2: $a \leftarrow min(e, \pi)$
3: $b \leftarrow max(e, \pi)$
4: **if** $sat(\pi \wedge (\wedge_{x \in range(M,a,b)} e \neq x.e))$ **then**
5: $\bar{t} \leftarrow \bar{t} - 1$
6: $M \leftarrow M \cup \{(a, b, e, new\_symbol, \bar{t}, true)\}$
7: **end if**
8: $v \leftarrow 0$
9: **for** $x \in range(M, a, b)$ by ascending timestamp **do**
10: $v \leftarrow ite(e = x.e \ \wedge \ x.\delta, \ x.v, \ v)$
11: **end for**
12: **return** v
13: **end function**

This optimization proves to be extremely effective in practice when dealing with real-world programs presenting broad pointers. Interestingly, a recent work [24] targeting KLEE has proposed one semantics-preserving transformation for array operations that shares the spirit of our optimization. This transformation targets constant arrays and can cope even with cases where the same value is referred by non-consecutive indices within an array. However, their rule does not take into account timestamps and thus cannot be adapted straightforwardly in the context of MEMSIGHT.

3.3. Discussion

Previous work [2] hinted that certain bugs can only be revealed when writes are modeled symbolically. However, classic approaches to memory modeling may not scale. For instance, Stephens *et al*. [25] remark that "a repeated read and write using the same symbolic index would result in a quadratic increase in symbolic constraints or [...] the complexity of the stored symbolic expressions". Our solution offers a more compact encoding that does not ask the solver to enumerate all matching addresses – a possibly expensive task [16] – while range intersection operations can be offloaded to efficient data structures such as the paged interval tree.

When considering our running example of Figure 1 in the context of the standard memory models of ANGR and KLEE the results of the symbolic exploration can differ compared to what we have seen with MEMSIGHT. ANGR adopts a partial model that would concretize both the symbolic write `a[i]` and the symbolic read `a[j]`, as the memory `&a[j]` spans is large. Even when calling-context information yields intervals of manageable size, the symbolic read `a[j]` would account for each `a+j` address instance individually, leading the executor to find only one bomb-defusing input, i.e., the one in which concrete address `a+j` equals the value chosen for `a+i` by the write concretization strategy. A fully symbolic approach—i.e., a memory model implemented as in ANGR but that avoids address concretizations by raising the read and write thresholds to an infinite value—would instead reveal the property that all inputs in which `i==j` holds defuse the bomb. Unfortunately, as we will show in Section 5, a fully symbolic memory as described above hardly scales in practice.

On the other hand, if the pointer to array `a` is concrete, then KLEE quickly finds the bomb-defusing input. However, if `a` is symbolic then KLEE needs to fork the execution for each active memory object, possibly leading to poor performance. Depending on the considered current execution state, this approach may or may not scale in practice. Nonetheless, a slightly more sophisticated variant of the running example can get KLEE in trouble again. Consider the code provided in Figure 6. Function `bomb2` is very similar to the spirit of function `bomb`, except for it handles a two-dimensional array in place of a one-dimensional array. In particular, it takes as inputs

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

```
1: void bomb2(char ** a, unsigned char i, unsigned char j, unsigned char k) {
2:     char boom;
3:     a[i][k] = 23;
4:     if (a[j][k] == 23) boom = 0;
5:     else               boom = 1;
6:     assert(!boom);
7: }
```

Figure 6. A more sophisticated variant of the running example presented in Figure 1.

a matrix `a`, which is assumed to be dynamically allocated, and three symbolic indices `i`, `j`, and `k`. Notice that, since the type of these indices is `unsigned char`, their values can range between 0 and 255. For the sake of simplicity, we assume that the matrix has been properly allocated, i.e., it contains at least 256 rows and at least 256 columns. Similarly to the running example, we assume that the content of each cell is initialized with the zero constant. In this scenario, the memory model used by KLEE leads it to generate 256×256 states. The write operation on line 3 would generate 256 states since `a[i]` may point to any of the rows in the matrix. Similarly, when exploring each of the 256 states, KLEE would again fork 256 states when considering the read access on line 4 since `a[j][k]` may point to any of the rows. This leads to as many as 65536 states. Although this is not a very large number and 256 of such states can actually make the assertion fail (i.e., for each state forked at line 3, there is one of its children at line 4 that satisfies the condition `i == j`), KLEE takes more than 2 hours to generate a valid solution in the experimental setup of Section 5.1.1. In contrast, by using MEMSIGHT in a symbolic executor that models memory as a single flat region as discussed in Section 4.1, its efficiency mainly depends on the number of bytes previously written in memory, allowing it to find a valid solution for this example in about 30 minutes.

4. IMPLEMENTATION

In this section, we present how we have integrated MEMSIGHT in two state-of-the-art symbolic execution frameworks: ANGR and KLEE. The first framework is a natural choice since one of the design goals behind our model is to support symbolic execution as a means for implementing exploitation techniques. ANGR has proved to be worth for this goal during the DARPA Cyber Grand Challenge, when the MechaPhish team used it to detect, patch, and exploit vulnerabilities in complex software. The second framework is instead one of the most robust and mature symbolic execution engines currently available to the research community. KLEE has been used in several academic as well as industrial projects, demonstrating its effectiveness even on large programs that cannot be currently analyzed with ANGR. KLEE ships with robust POSIX environment models and works on the LLVM IR of a program which preserves some structural properties about the program, making the symbolic execution simpler: for instance, the IR maintains the number of elements of statically allocated arrays and the number of fields in a C structure along with their type. By integrating our model into KLEE, we have the opportunity to compare our approach, that only requires the theory of bitvectors, to a memory model that intrinsically depends on the theory of arrays.

4.1. MEMSIGHT *in* ANGR

We start by providing the reader with an overview of the ANGR framework, and then describe how we have implemented MEMSIGHT within its symbolic execution engine.

ANGR. ANGR is an open-source framework, mainly written in Python, for performing binary analysis. In Figure 7a we provide a simplified overview on the major components that constitute ANGR. Binary code is lifted from machine code to VEX IR using the *PyVEX* package. The symbolic execution engine is contained in the *angr* package and evaluates the VEX IR of the program under analysis, constructing expressions over symbolic variables that can be processed by the Z3 SMT solver using the *claripy* package. Finally, the *CLE* and *archinfo* packages abstract away several

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

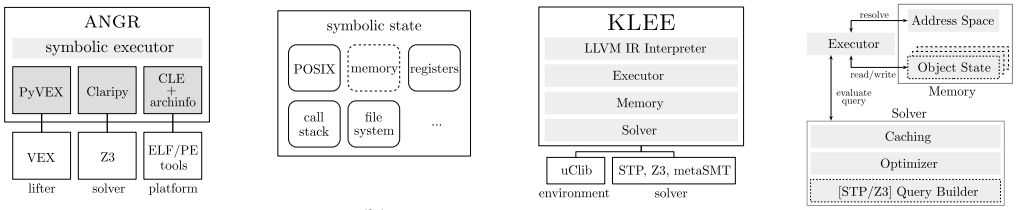

Figure 7. ANGR: (a) framework overview and (b) symbolic state and its plugins. KLEE: (c) framework overview and (d) interactions between the executor, the memory, and the solver layers.

low-level details of the hardware architecture and the operating system on which the binary under analysis is supposed to run.

As discussed in Section 2.2.1, ANGR adopts a partial memory model by constructing an ITE expression for a symbolic read if the spanned range is not too large, querying Z3 for the maximum and minimum values for the address. If they differ at most by 1024, ANGR will ask Z3 to enumerate all the solutions and build an ITE accordingly, otherwise it will concretize the address. Optionally, also write accesses can be treated as symbolic, with a suggested threshold of 128 for range size.

The *claripy* constraint-solving wrapper implements a number of optimizations to relieve the solver from the burden of repeated queries and improve efficiency. For scalability ANGR implements an extended *veritesting* merging strategy [4] that analyzes the control flow graph to determine places where is profitable to condense the effects of separate chunks of code using ITE constraints.

MEMSIGHT Integration. In ANGR, the symbolic state is made of different components dubbed plugins (see Figure 7b). Each plugin takes care of one aspect of the execution state. For instance, the *POSIX* plugin provides support for several interaction mechanisms available on environments conforming to the POSIX standards, for instance for networking and file-system operations. To integrate MEMSIGHT into ANGR, we have devised it as a plugin* implementing the memory abstraction required by the symbolic engine of ANGR, so it can easily be interchanged with the default plugin for partial memory modeling. As the abstraction explicitly accounts for a merging primitive, strategies such as veritesting can run on top of MEMSIGHT with no extra effort required.

4.2. MEMSIGHT *in* KLEE

As for ANGR, we first present an overview of the KLEE framework, and then describe how we have integrated MEMSIGHT within this symbolic execution engine.

KLEE. KLEE is an open-source symbolic virtual machine written in C++ that is able to evaluate programs expressed in LLVM IR, which is typically obtained during the lowering phase when compiling C, C++, and other high-level languages supported by the LLVM compiler. A simplified overview of KLEE is provided in Figure 7c, where several key software layers are depicted. To symbolically analyze a program, the executor evaluates the program through an LLVM IR interpreter. When memory operations are met, the executor interacts with the memory layer. This component (Figure 7d) is composed of two main abstractions. The *Address Space* is used by KLEE to determine (*resolve* in KLEE terminology) which memory objects may be impacted by a memory access. As discussed in Section 2.2.2, KLEE queries the solver to check whether a symbolic pointer can fall within one or more memory objects that are active in the address space of the program. For each memory object that could be affected, KLEE forks the execution and performs the load or store operation within each state through the *Object State* instance associated to the memory object of interest. The Object State abstraction takes care of constructing expressions that can later be translated by KLEE into array expressions. When the executor needs to interact with the solver, e.g.,

*Our prototype was tested in ANGR v5.6 and is available at: `https://github.com/season-lab/memsight`. Creating it has required a substantial implementation effort due to its complex interplay with the different layers of ANGR. Along the way, we have discovered and fixed a few subtle bugs in ANGR.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

to check the feasibility of a branch, it submits a query to the solver abstraction. This component is composed of several software layers: the caching layer is used to reuse assignments obtained during previous queries; the optimizer is used to simplify and rewrite expressions; the query builder is the actual solver backend that translates a query from the KLEE format to the solver-specific format, e.g., the Z3 builder constructs a query through the API exposed by the Z3 framework.

MEMSIGHT Integration. To integrate MEMSIGHT in KLEE, we have explored two different approaches. Initially, we have developed a custom implementation of the Object State abstraction. This choice closely resembles the implementation strategy adopted in the ANGR framework. However, two drawbacks soon emerged. First, our custom implementation of the Object State constructs ITE expression rather than array expressions. KLEE expects to materialize array items, while in our case it has to materialize bitvectors. Although items of arrays in KLEE are in turn bitvectors and thus this difference should not pose a severe impact, we had to intervene inside several KLEE internals to make MEMSIGHT works as expected. Another, and possibly more important, source of concern was related to how KLEE rewrites and optimizes array expressions, reducing often by a significant extent the need to interact with the solver. By constructing ITE expression rather than array expressions, our first implementation of MEMSIGHT lost the possibility to benefit from these optimizations and rewriting rules added to KLEE over the years. Although several such optimizations could be rewritten to work on ITE expressions, the implementation effort required to this end was a daunting prospect.

For these reasons, we have explored a different implementation approach for integrating MEMSIGHT into KLEE: we devised MEMSIGHT as a rewriting rule within the solver-specific query builder component. In particular, we altered the Z3 and the STP builder in order to convert array expressions generated by the standard implementation of the Object State into ITE expressions. Since an array expression is expressed as a `STORE` expression that recursively describes the history of the write operations performed on an array from its initial state, we were able to generate ITE expressions for our setting by replaying these store operations on a bitvector instead of an array[†]. Following this strategy, our changes impacted the internals of KLEE only for the lower layers of the solver abstraction. In this way, we benefit from any optimizations that happen at higher layers of the solver abstraction and more in general in the whole framework. A downside of this implementation approach – which emerges in the experimental evaluation discussed in Section 5.2 – is that we have added an additional rewriting pass to the builder, possibly increasing the runtime overhead for this layer. In the remainder of this article, we only consider this second approach when discussing the integration of MEMSIGHT in KLEE. This choice arises from experimental results on different benchmarks that clearly showed how the performance of our first implementation strategy can be severely harmed by the lack of several optimization and rewriting rules shipped with the standard KLEE framework.

One relevant consideration about our implementation strategy is that we did not alter the Address Space abstraction. In other words, our solution is affected by one of the drawbacks from the memory model adopted by KLEE: whenever a memory access may affect different objects, our customized variant of KLEE will still fork the execution state. There are two motivations behind our choice. First, in order to avoid forks we should have altered large, significant portions of KLEE, and in light of the many features hinging on them that were added over the years we were concerned that the robustness of the exploration could have been affected. Second, by altering the memory model of KLEE we would have lost the ability to perform a fair yet simple and meaningful comparisons between the standard release of KLEE and our variant. Our integration strategy preserves how KLEE reasons on the different memory objects active within the program execution, and only alters how KLEE reasons within each memory object: standard KLEE exploits the theory of arrays, while MEMSIGHT only requires the theory of bitvectors.

[†]For instance, the array expression `v = SELECT(STORE(STORE(ARRAY(size=2), 0, 1), 1, 2), `α_x`)`. where α_x is a symbolic input, is converted into the following replay sequence: `bv = MemSight(size=2); bv.store(0, 1); bv.store(1, 2); v = bv.load(`α_x`);`

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

5. EVALUATION

Due to our choice of integrating MEMSIGHT into two symbolic executors that target different application scenarios, the experimental evaluation of MEMSIGHT has been split into two parts. Before we move to the presentation of the results, we would like to provide the reader with some considerations on the scope of the two engines that motivated our choice.

Application Domains. The ability of ANGR to reason on binary code lifted to platform-agnostic VEX IR makes it an extremely powerful, versatile tool for security researchers that exploit symbolic execution with the ultimate goal of, e.g., finding vulnerabilities [5], bypassing authentication checks in device firmware [26], or dissecting malware [27]. Although ANGR integrates several state-of-the-art program analysis techniques such as veritesting [4] to improve scalability, it still struggles when analyzing real-world programs. Also, the lack of robust environment models in ANGR makes it really challenging to use it for software testing, even on standard UNIX tools. For instance, when analyzing the `coreutils` suite with ANGR we observed a large number of false positives (i.e., unrealistic program behaviors) that make it hard to perform meaningful comparisons with results obtained from other executors that use different memory models. Nonetheless, ANGR has improved drastically during the last years and thus we believe that the situation can improve in a near future. In light of these considerations, we focused our investigation with ANGR on binaries taken from DARPA Cyber Grand Challenge (CGC): ANGR provides a robust support to the DECREE execution environment adopted during this competition, allowing us to analyze several complex pieces of code written by DARPA for this platform.

On the other hand, KLEE has been designed with the ultimate goal of performing software testing on real-world programs. To this end, its creators have built robust environment models that make it possible to use KLEE for testing programs written for POSIX platforms. However, KLEE cannot be used directly for analyzing programs whose source code is not available, and it is not suitable for exploiting, or even just finding, software vulnerabilities whose nature is strictly related to the binary representation of a program. For instance, analyzing the consequences of a stack buffer overflow can be quite challenging when the order of stack variables has not yet been fixed by the compiler.

5.1. ANGR

In this section we report on an investigation of the practical impact of MEMSIGHT when integrated inside ANGR. The main goal of our discussion is to answer the following two research questions:

RQ1 How broad are the symbolic addresses used in benchmarks taken from the DARPA CGC?

RQ2 Does MEMSIGHT improve ANGR's soundness?

5.1.1. Experimental Setup and Methodology Our experiments are based on benchmarks authored by the Cromulence (CROMU) DARPA performer group of the CGC. Although these programs can hardly be considered *real-world*, since they can run only within the DARPA Experimental Cybersecurity Research Evaluation Environment (DECREE)*, they have been designed by DARPA to resemble the complexity emerging in real-world applications. Additionally, DARPA has put significant effort in order to design applications that can effectively pose severe challenges to state-of-the-art program analysis techniques, including symbolic execution.

Tests were conducted on a server equipped with an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz with 16 cores and 64 GB RAM, running Linux CentOS 6.7. We compare MEMSIGHT against three different ANGR memory concretization strategies: 1) ANGRCONC, which concretizes all accessed symbolic addresses, 2) ANGRPART (the default of ANGR), which concretizes all read addresses that span ranges larger than 1024 and all written addresses, and 3) ANGRFULL, which avoids address concretizations by raising both the read and write thresholds for concretization to an infinite value.

*DECREE is an open source operating system extension based on a custom x86 Linux kernel that supports only 7 system calls. Its simplicity removes several implementation details of standard UNIX-based operating systems, making DECREE ideal for computer security research.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

We symbolically execute MEMSIGHT, ANGRCONC, ANGRPART, and ANGRFULL on the CROMU benchmarks with a budget of 2 hours and 32 GB of RAM. To characterize the breadth of the exploration enabled by the memory model in use, we measure the number of explored paths. To make a meaningful comparison across different memory models, in our experiments symbolic execution explores states in rounds using a first-in-first-out strategy: a new round starts only when all states in the previous round have been explored. Hence, at any round the set of explored paths with some concretization strategy is always a subset of the paths that would be explored by performing fewer concretizations. To minimize the memory consumption of ANGR, we enabled veritesting [4]. As a result, ANGR may symbolically execute multiple basic blocks, i.e., straight-line code sequences with no branches, from the same state within a single round. Additionally, ANGR may opt to merge some states before ending a round. For this reason, the round count should be considered just as an indirect measure of the number of instructions explored during an experiment.

5.1.2. Experiments Table I provides an overview of several statistics that we collected when analyzing a representative selection of benchmarks from the CROMU corpus. In particular, to shed light on our first research question (RQ1) we selected from the CROMU set a total number of 20 programs, dividing them into three groups. The first group (rows 1-3) contains benchmarks where we observed extremely broad pointers that lead to concretization by ANGRPART (the default memory model in ANGR). The second group (rows 4-10) contains programs that also show concretization events when using ANGRPART. Lastly, the third group (rows 11-20) contains benchmarks where symbolic pointers either have not been observed or were not broad enough to require concretization when using ANGRPART. Benchmarks have been selected to reflect the ratios observed in preliminary analyses on the full CROMU set, with a large fraction of benchmarks from the CROMU corpus not showing symbolic pointers that would be concretized by ANGRPART, while only a few presented extremely broad pointers. Unfortunately, we were unable to run several CROMU benchmarks due to internal assertion failures or errors in ANGR, so we could not obtain reliable information on the characteristics of their memory accesses.

Columns *# symbolic pointers* and *max range size* report the total number of memory accesses to a symbolic address with a range size larger than 1 and the maximum size of the ranges of symbolic addresses accessed by load and store operations throughout the execution. Notice that for benchmarks in the first group the ranges are much larger than the thresholds one can afford in practice when using partial memory models. Column *expected outcome of the exploration* speculates on what we may expect from symbolically executing these benchmarks with different memory models. Our hypotheses are based on two considerations. First, extremely broad pointers should demonstrate how a fully symbolic memory approach as in ANGRFULL cannot scale in practice. Second, symbolic accesses that are concretized may lead an executor to miss some paths during the exploration. Notice that this is not obvious, as such a path can be missed by an executor only when the result of a pointer concretization affects the decision taken on some branch condition along it.

To address our second research question (RQ2), we first assess to what extent concretization may restrict state explorations, possibly leading to unsoundness due to missed paths. Table II presents a summary of our experimental comparison; its left part compares the number of distinct control flow paths explored by the memory models we considered. To allow for a direct comparison, the snapshot of the number of paths is taken at the same exploration round K for the same benchmark in all memory versions. The value chosen for K is the same considered in Table I. We first observe that full concretization (ANGRCONC) may significantly restrict the number of explorable paths, confirming the findings reported in previous works [2, 16]. For instance, as a consequence of concretizing symbolic pointers on the `CROMU_00003` benchmark (which features symbolic loads) we lose 3566 out of 3592 paths under ANGRCONC. Overall, when considering our third group of benchmarks (rows 11-20), half of them generates more paths when not performing concretization. The increase of paths is observable on every benchmark of this group with at least one symbolic pointer.

When taking into account benchmarks from the second group (rows 4-10), we note that a partial memory model (ANGRPART) does not often capture all explorable paths. Interestingly, even a few symbolic write operations spanning small memory regions can lead programs to generate a

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table I. Statistics on the number of concretization events and broadness (number of possibly dereferenced distinct bytes) of symbolic pointers when considering the CROMU Cyber Grand Challenge benchmarks.

#	BENCHMARK	ROUND K	# SYMBOLIC POINTERS	MAX RANGE SIZE		EXPECTED OUTCOME OF THE EXPLORATION
				LOAD	STORE	
1	CROMU_00006	316	6	24	262128	ANGRCONC and ANGRPART may miss some paths
2	CROMU_00018	923	716	400	770048	ANGRFULL may take too long
3	CROMU_00038	389	7	4	65536	MEMSIGHT should explore all paths
4	CROMU_00001	481	235	0	9	ANGRCONC and ANGRPART may miss some paths ANGRFULL and MEMSIGHT should explore all paths
5	CROMU_00009	1534	662	8	9	
6	CROMU_00014	1511	1521	0	9	
7	CROMU_00024	329	1708	28	28	
8	CROMU_00031	899	126	2295	56	
9	CROMU_00032	643	52	576	8192	
10	CROMU_00033	364	1053	1020	1020	
11	CROMU_00002	773	0	0	0	ANGRCONC may miss some paths all paths should be explored by ANGRPART, ANGRFULL, and MEMSIGHT
12	CROMU_00003	646	52	24	0	
13	CROMU_00004	9342	0	0	0	
14	CROMU_00005	79754	0	0	0	
15	CROMU_00008	9540	0	0	0	
16	CROMU_00012	365	72	24	0	
17	CROMU_00022	545	33	28	0	
18	CROMU_00029	489	469	12	0	
19	CROMU_00040	946	396	16	0	
20	CROMU_00041	840	0	0	0	

Table II. Experimental comparison between ANGRCONC, ANGRPART, ANGRFULL, and MEMSIGHT when considering the CROMU CGC benchmarks. Round K refers to the exploration depth reported in Table I.

BENCHMARK	AT ROUND K											
	# PATHS				TIME				MEMORY			
					SECS	SLOWDOWN			MB	OVERHEAD		
	ANGR CONC	ANGR PART	ANGR FULL	MEMSIGHT	ANGR CONC	ANGR PART	ANGR FULL	MEMSIGHT	ANGR CONC	ANGR PART	ANGR FULL	MEMSIGHT
CROMU_00006	9	31	**fail**	**34**	56	1.8	**fail**	10.9	222	2.0	**fail**	4.8
CROMU_00018	78	96	**fail**	**383**	294	1.0	**fail**	20.7	1195	1.0	**fail**	12.2
CROMU_00038	5	5	**fail**	**127**	314	0.9	**fail**	21.8	315	1.1	**fail**	42.7
CROMU_00001	1845	1845	**2466**	**2466**	2084	1.0	1.2	1.4	23821	1.0	1.3	1.3
CROMU_00009	1131	1131	**1715**	**1715**	1729	1.1	1.7	1.9	11493	1.0	2.5	2.6
CROMU_00014	2072	2072	**2128**	**2128**	4244	1.0	1.1	1.2	14213	1.0	1.2	2.0
CROMU_00024	2100	2100	**2673**	**2673**	1216	1.0	3.5	4.8	18252	1.0	1.4	1.8
CROMU_00031	299	1413	1413	1413	841	3.6	3.5	4.3	5272	5.7	5.8	5.9
CROMU_00032	3	3	**53**	**53**	57	1.0	**119.9**	**11.6**	160	1.0	28.5	21.7
CROMU_00033	508	508	**539**	**539**	1765	1.0	1.4	1.3	26110	1.0	1.1	1.2
CROMU_00002	2468	2468	2468	2468	1408	1.1	1.0	1.1	19547	1.2	1.0	1.0
CROMU_00003	26	3592	3592	3592	95	31.9	33.2	35.4	359	67.3	63.0	68.9
CROMU_00004	1	1	1	1	7194	0.9	1.0	1.0	689	1.0	1.0	1.6
CROMU_00005	1	1	1	1	6592	1.0	1.0	1.1	2378	1.0	1.0	1.0
CROMU_00008	1	1	1	1	6811	1.0	1.0	1.1	722	1.0	1.0	1.5
CROMU_00012	3524	3740	3740	3740	2463	1.1	1.2	1.5	24996	0.9	1.0	0.8
CROMU_00022	1063	1156	1156	1156	2425	1.2	1.2	1.3	16725	1.0	1.1	0.9
CROMU_00029	2142	3549	3549	3549	2461	1.6	1.7	1.8	18270	1.4	1.2	1.3
CROMU_00040	1322	2882	2882	2882	2982	2.2	2.2	2.3	16358	1.0	1.0	1.2
CROMU_00041	1316	1316	1316	1316	6569	1.0	1.0	1.1	14611	0.7	1.0	0.7

large amount of additional execution paths. For instance, ANGRPART misses more than 33% (621 out of 2466) of the paths in CROMU_0001 due to concretizations on write accesses spanning at most 9 bytes. Nonetheless, on one benchmark (CROMU_00031) we can see how despite some concretizations take place when using ANGRPART, no additional paths could be observed in the experiment with ANGRFULL.

Benchmarks from the second group seem to suggest ANGR is missing several paths due to concretization. MEMSIGHT is instead able to explore the same number of paths as ANGRFULL. However, it is only when considering benchmarks presenting extremely broad pointers (first group, rows 1-3) that we can appreciate the benefits given by MEMSIGHT, completing our discussion of research question (RQ2). ANGRFULL cannot handle in practice pointers spanning large memory regions, failing due to excessive resource requirements. In particular, even a few large write operations were enough in our experiments to make ANGRFULL exhaust its time budget. On the other hand, MEMSIGHT is the only memory model among the ones considered in our evaluation that lets ANGR fully explore paths in the presence of extremely broad pointers.

The benefits from avoiding pointer concretization come however with a price. The right part of Table II reports the slowdown (running time) and the spatial overhead (memory usage) measured on the experiments with ANGRPART, ANGRFULL, and MEMSIGHT compared to ANGRCONC. The attentive reader would argue that observing consistently higher running time and memory overhead is however expected when additional paths emerge during the exploration as a consequence of avoiding concretizations. For instance, we measured a slowdown higher than $30\times$ and a spatial

Copyright © 2019 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

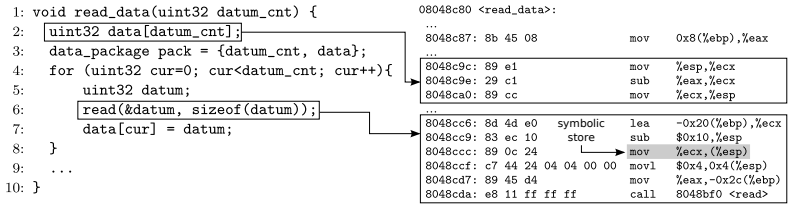

Figure 8. Example of symbolic store excerpted from the CROMU_00006 CGC benchmark.

overhead higher than 60× on the CROMU_00003 benchmark where ANGRCONC has missed the majority (99,3%) of paths during its exploration.

The most interesting performance comparison that we can make from the data reported in Table II is between ANGRFULL and MEMSIGHT. In presence of symbolic pointers spanning small memory regions, ANGRFULL is generally more efficient than MEMSIGHT. We were not surprised since MEMSIGHT maintains additional metadata (e.g., timestamps) compared to ANGRFULL: their storage and processing can be quite expensive in terms of memory and time usage. However, MEMSIGHT shows its advantages when symbolic pointers become broader. For instance, CROMU_00032 performs a few write operations with a range equal to 8192 bytes and MEMSIGHT is considerably more efficient than ANGRFULL on it. When considering benchmarks with extremely broad pointers (rows 1-3) MEMSIGHT is the only model that keeps their exploration sustainable in ANGR.

5.1.3. Case Study We now discuss a real-world code example that shows how MEMSIGHT can maintain fully symbolic addresses in a context where previous techniques are instead forced to concretize them. CROMU_00006 is a service that produces random numbers and generates charts for numeric data, including bar charts and sparklines. As seen in Table I, this benchmark dereferences pointers that span very large portions of the address space. This arises for instance in function `read_data` shown in Figure 8, which fills a buffer `data` of symbolic size `datum_cnt` with values read from the input. An inspection of the x86 binary code reveals that the dynamic stack allocation `uint32 data[datum_cnt]` (line 2) makes the stack pointer register `esp` symbolic. Later in the code (line 6), parameter passing on stack to function `read` results into a symbolic store via `esp`. The range of possible addresses `esp` can assume at that point is as large as 262,128 due to previous constraints on the maximum stack size imposed by the program. This triggers concretization in ANGRPART, forcing the symbolic execution to reason on a buffer of fixed size[†]. Since the symbolic range for `esp` is very large, ANGRFULL fails to produce a result due to excessive resource consumption. In contrast, MEMSIGHT keeps `esp` symbolic, considering in the analysis all possible sizes of the `data` buffer. As confirmed by our experiments (Table II), the ability to consider a buffer of variable size impacts the breadth of the exploration, allowing MEMSIGHT to push symbolic execution through states that remain hidden to ANGRPART.

5.2. KLEE

We now analyze how our integration of MEMSIGHT compares with the standard release of KLEE. As discussed in Section 4, our implementation in KLEE does not affect the memory layout reasoning adopted in this framework, but impacts how KLEE behaves when reasoning over a memory object. Since KLEE uses theory of arrays to accurately support memory object reasoning, concretization on symbolic accesses does not play a role as seen with ANGR. For this reason, differently from Section 5.1.2, we cannot expect MEMSIGHT to explore additional paths when compared to the standard KLEE. On the other hand, our evaluation aims at answering the following questions:

RQ1 How frequent are symbolic accesses when considering real-world programs?

[†]We observed that, since the stack grows downward and ANGR concretizes symbolic writes by default using the maximum possible address, then the analysis ends up reasoning on the smallest, rather than the largest buffer size. A segment-dependent concretization strategy would yield better results here.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

RQ2 How efficient is KLEE when using MEMSIGHT to reason over a memory object?

RQ3 How can differences in performance be explained?

5.2.1. Experimental Setup and Methodology To make a fair comparison between MEMSIGHT and the standard memory model of KLEE, we replicated as closely as possible the experimental setup used in a recent work targeting the KLEE framework [24]. This operation was favored by the availability of the artifact[§] associated to the original paper. In our experiments, we considered the following benchmarks:

- `bandicoot` v6, a programming system with a set-based programming language;
- `coreutils` v6.11, a collection of programs for file, shell, and text manipulation;
- `bc` v1.06, an arbitrary-precision calculator that solves mathematical expressions.

Tests were conducted on a server equipped with two Intel® Xeon® E5-4610 v2 @ 2.30GHz with 8 cores each and 256 GB RAM, running Debian Linux 9.2. We built KLEE from the same source code base released with the artifact, adding our patches when running KLEE with MEMSIGHT. KLEE was executed matching the configuration *combined transformation* originally proposed and tested in the paper [24]. We used STP 2.1.2 as the constraint solver [17].

To validate our experiments, as for the setup, we pursued the same strategy proposed in the artifact [24]. In particular, we executed programs using a DFS search heuristic, a deterministic allocator, and a fixed sandbox for file system operations. To make executions across different memory models comparable, we fixed a number of target instructions that must be explored by each program when running under KLEE. Our targets derive from the ones proposed in the original artifact [24] with adjustments due to a different hardware configuration (e.g., a different CPU). We report our targets in Table III (column *instructions*). To ensure that different runs of the same benchmark executed the *same* instruction trace, we exploited the capabilities of KLEE to record an instruction trace and then compared the traces across different runs to identify possible divergences. A divergence could indicate a bug in our implementation resulting in a different program behavior.

5.2.2. Experiments Table III provides a summary of our experiments. The columns grouped under *statistics* report how many load and store symbolic accesses have been detected when executing the considered set of programs under the baseline KLEE. The numbers should help answer our first research question (RQ1). Besides counting the number of accesses, as for the experiments with ANGR, we measured how large were the memory regions spanned by each symbolic access, reporting in Table III the maximum value observed during an experiment. Notice that, differently from the experiments with ANGR, we considered an access as symbolic even when it spans a single memory location since KLEE does not check for its range and thus treats it as symbolic.

The majority of the programs that we considered show some symbolic accesses during their execution. When considering load operations (column *symbolic load*), we can see that most of them performs a non-negligible number of accesses using pointers that are not concrete. However, only few programs (namely `bc`, `csplit`, `paste`, and `tsort`) have at least one symbolic load operation whose range is larger than 1024 (i.e., a threshold commonly used by partial memory models). When considering instead symbolic write operations (column *symbolic write*), only 8 out of 46 programs have at least a symbolic access. However, just half of them has at least one symbolic write operation whose range is larger than one. Overall, symbolic accesses spanning large memory regions do not appear to be common in the set of benchmarks that we considered, with only few notable exceptions.

The columns grouped under *comparison* in Table III provide the results of a performance comparison between the standard memory model of KLEE and MEMSIGHT, shedding light on our second research question (RQ2). For each program and for each memory model, we repeated the experiment five times, taking the average running time and computing a 95% confidence interval. By comparing the instruction traces recorded during different runs of the same program, we verified that no divergence occurred, allowing us to make a meaningful comparison among runs. To make running times easier to compare, column *speedup* reports the ratio between the average running time when using KLEE with its standard memory model and the average running time with KLEE when

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table III. Comparison between KLEE and MEMSIGHT when running the same instructions from the same programs. Highlighted benchmarks are considered by charts available in Figure 9.

BENCHMARK	STATISTICS						COMPARISON		
	SYMBOLIC LOAD		SYMBOLIC STORE		# INSTR	QUERY TIME	RUNNING TIME (SECS)		SPEEDUP
	#	MAX RANGE	#	MAX RANGE			KLEE	MEMSIGHT	
[	760	745	0	0	2062048	72.0%	36.80 ± 1.28	36.80 ± 1.28	1.00
ar	304	1021	0	0	18456	96.0%	862.20 ± 8.00	720.80 ± 7.31	**1.20**
as	1304	1021	0	0	3488967904	1.0%	2605.80 ± 134.70	2612.80 ± 125.47	1.00
bandicoot	903615	256	0	0	17122168	<1%	1110.40 ± 5.86	1137.00 ± 3.18	0.98
base64	91866	511	331	1	1034873341	1.0%	1519.40 ± 37.17	1519.80 ± 58.61	1.00
basename	272	509	0	0	20794661	9.0%	28.60 ± 0.54	29.20 ± 1.07	0.98
bc	658	1151	0	0	267080	90.0%	1226.20 ± 36.85	973.80 ± 12.01	**1.26**
comm	681	509	0	0	951360500	12.0%	1477.40 ± 38.08	1476.20 ± 31.19	1.00
csplit	10257	2041	0	0	355387987	4.0%	2842.80 ± 21.49	2840.40 ± 24.68	1.00
cxxfilt	14489	509	5233	1	70769782	87.0%	3163.60 ± 198.55	2710.00 ± 110.71	**1.17**
dircolors	673	511	0	0	740772634	1.0%	1426.00 ± 22.04	1399.00 ± 15.12	1.02
dirname	952	509	0	0	14097923	13.0%	22.20 ± 0.82	21.80 ± 0.82	1.02
elfedit	7563	509	0	0	2146961	14.0%	1277.00 ± 3.53	1275.60 ± 1.78	1.00
env	0	0	4610	2	3796866887	<1%	3254.40 ± 160.73	3259.00 ± 114.89	1.00
expand	209359	511	0	0	1188404283	17.0%	2801.20 ± 48.57	2805.60 ± 42.32	1.00
expr	12126	19	0	0	36734777	81.0%	211.20 ± 2.12	220.20 ± 1.89	0.96
factor	14866	511	0	0	209953760	33.0%	508.40 ± 6.88	514.40 ± 10.77	0.99
fmt	5666	511	0	0	123867149	5.0%	463.40 ± 2.36	464.60 ± 4.42	1.00
fold	0	0	0	0	1298980327	<1%	1640.00 ± 63.49	1600.80 ± 29.26	1.02
id	274	509	0	0	3312410730	<1%	2921.40 ± 104.60	2701.60 ± 59.29	1.08
join	190164	511	0	0	2050775816	7.0%	2906.20 ± 53.17	2786.00 ± 79.42	1.04
link	62500	509	0	0	352800856	<1%	1374.80 ± 9.26	1374.40 ± 12.86	1.00
ln	66590	509	15	1	1560875063	1.0%	1768.40 ± 37.80	1784.00 ± 25.70	0.99
logname	9258	509	0	0	113008072	<1%	1679.20 ± 24.84	1668.20 ± 9.56	1.01
mkfifo	11571	509	0	0	173858572	1.0%	268.00 ± 9.19	269.60 ± 4.42	0.99
mknod	15235	509	0	0	177406898	32.0%	644.20 ± 11.07	625.40 ± 5.99	1.03
nice	8674	511	0	0	31381687	88.0%	1874.00 ± 59.53	2925.40 ± 45.52	**0.64**
nm-new	160	511	3011	97	410382622	43.0%	507.80 ± 8.00	508.20 ± 15.61	1.00
objcopy	718	511	5745	97	1201202143	63.0%	2504.20 ± 127.30	2371.00 ± 162.54	1.06
od	6358	511	63	1	112411438	68.0%	832.20 ± 17.01	861.40 ± 14.63	0.97
paste	460457	225001	0	0	1739528644	<1%	2968.60 ± 62.48	2945.20 ± 71.82	1.01
readelf	7470	509	0	0	2290974	15.0%	1212.20 ± 5.34	1214.20 ± 5.30	1.00
readlink	274	509	0	0	1691286445	<1%	1885.40 ± 72.24	1859.40 ± 65.24	1.01
rmdir	27327	509	0	0	335183764	<1%	517.00 ± 6.08	522.40 ± 11.29	0.99
seq	7491	509	216	1	44348415	87.0%	1252.20 ± 39.97	852.80 ± 12.31	**1.47**
setuidgid	9573	511	0	0	120234190	1.0%	1655.40 ± 14.25	1670.00 ± 7.84	0.99
sleep	24443	509	0	0	292909689	1.0%	496.80 ± 9.56	510.00 ± 9.37	0.97
strip-new	329	511	3043	97	458212770	54.0%	784.80 ± 48.87	629.00 ± 52.91	**1.25**
stty	320569	511	0	0	598834807	1.0%	2943.00 ± 35.25	2883.60 ± 14.92	1.02
tr	15399	509	3869	255	89174737	41.0%	2062.80 ± 8.36	1478.80 ± 11.25	**1.39**
tsort	2345	140733033363337	0	0	6001673	94.0%	1137.80 ± 8.27	647.80 ± 9.83	**1.76**
tty	3143	509	0	0	740978348	1.0%	1418.40 ± 29.14	1403.60 ± 25.11	1.01
unexpand	65320	511	0	0	466186749	2.0%	946.00 ± 12.53	947.20 ± 11.16	1.00
users	1778	509	0	0	21790095	10.0%	32.00 ± 0.98	32.00 ± 0.98	1.00
wc	198146	511	0	0	904608440	<1%	1431.20 ± 48.93	1414.20 ± 19.44	1.01
whoami	9213	509	0	0	112545932	<1%	1624.60 ± 17.73	1627.40 ± 14.71	1.00

using MEMSIGHT. A speedup larger than 1.0 means that MEMSIGHT made the execution under KLEE faster, while a speedup smaller than 1.0 indicates a slower execution with MEMSIGHT with respect to the standard memory model of KLEE.

Only 8 out of 46 programs show a speedup that significantly deviates from 1.0 (bold values in the last column of Table III). The largest speedup (1.76) has been measured on `tsort`. Interestingly, this is the program where we observed the pointer spanning the largest memory region (140,733,033,363,337 bytes) across all the benchmarks. Similarly, the third largest speedup has been measured on the program `tr` that performs several store operations using pointers spanning a memory region of size 255 bytes, which is the largest one measured in our experiments when considering write accesses. In the same way, the positive performance impacts observed on the programs `ar`, `bc`, `cxxfilt`, `seq`, and `strip-new` may result from the symbolic accesses that take place during their executions. On the other hand, the `nice` program appear to be negatively affected by MEMSIGHT, resulting in a speedup equal to 0.64, i.e., a 36% slowdown during its execution. Other apparently unexpected results emerge also when considering benchmarks that perform symbolic accesses but do not seem to benefit from the use of MEMSIGHT.

To explain these results – thus tackling our third research question (RQ3) – we have to consider other factors that play a crucial role in the running time of a program when analyzed symbolically. A first key element is the time spent by KLEE querying the constraint solver during an experiment. If this time is rather small, then even in presence of symbolic accesses we cannot expect MEMSIGHT to impact the performance of KLEE significantly. Our implementation of MEMSIGHT inside KLEE works by rewriting queries submitted to the solver. Column *query time* in Table III provides an overview of the cumulative query time measured during our experiments when using the baseline KLEE. 30 out of 46 programs spent less than 20% of their running time querying the constraint solver

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

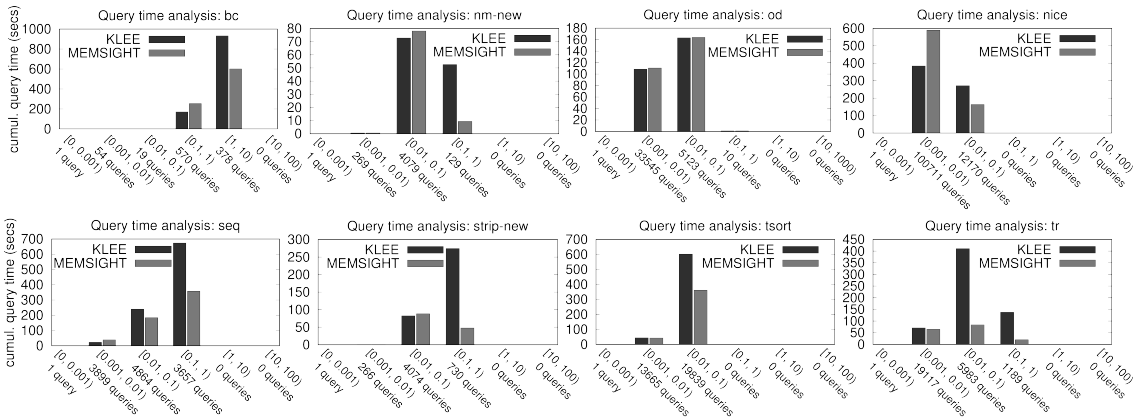

Figure 9. Analysis of the running time required to run queries generated by KLEE when symbolically analyzing programs highlighted in Table III.

when running under KLEE. For instance, although we observed a large number of accesses spanning large memory regions in `paste`, the query time during the experiment involving this program has been less than 1%, making really hard for MEMSIGHT to make any practical difference.

Taking into account the cumulative time spent querying the solver is not enough to explain the results for some of the benchmarks. For this reason, we decided to perform an additional experiment, where we focused our analysis on the time required for executing each specific query generated by KLEE. Unfortunately, we could not just measure the running time of each query during our experiments and then compare their running times when using different memory models. Since MEMSIGHT is generating queries for the backend solver that may differ from the ones generated by the standard memory model of KLEE, during an experiment we could get slightly different yet valid solutions depending on the memory model in use. Since the caching layer of KLEE may reuse solutions from previous queries to solve subsequent queries, several queries recorded with one memory model were not submitted to the solver when using another memory model, making any kind of comparison troublesome. Nonetheless, disabling the caching layer would have presented unrealistic performance results that could not be experienced in practice by any real user adopting a reasonable configuration for KLEE.

For this reason, we opted to execute KLEE with the caching layer enabled when generating results in Table III, and then perform an offline running-time analysis of the queries. In particular, we first recorded the queries submitted to the solver when running KLEE with its standard memory model. Given this fixed set of queries, we then used `kleaver` (a tool shipped with KLEE) to rerun them in isolation disabling any caching layer and reducing other runtime factors that could affect their execution when performed under KLEE. We repeated each query 10 times, measuring the average running time when using the standard STP solver backend of KLEE and then using MEMSIGHT. For the sake of simplicity, in the remainder of this section we refer to the standard STP backend of KLEE using the name KLEE, while MEMSIGHT is used to refer to the STP backend based on MEMSIGHT. Figure 9 provides the results of our analysis for the 8 programs highlighted in Table III. For each program, we partition the queries into six bins depending on their average running time when executed under KLEE. For each bin we report cumulative running times for the queries that fall into it when executed with KLEE, and for the same queries when executed with MEMSIGHT. The number of queries falling within each bin is shown under its label (x axis).

Overall, a trend seems to emerge in most charts: MEMSIGHT provides a positive performance impact for queries falling inside the partitions associated to larger query times, while the baseline KLEE is more efficient when we consider bins related to faster queries. Let us consider the `strip-new` benchmark: the 730 queries falling in the partition [0.1, 1) seconds are faster when using MEMSIGHT instead of KLEE; on the other hand, the 4074 queries from partition [0.01, 0.1) are slower with MEMSIGHT than with KLEE. Nonetheless, if we sum the cumulative times among the partitions we may understand why a 1.25 speedup was observed for `strip-new` in Table III. Observe that in doing so we might miss possible benefits from some optimizations in the caching layer, for instance when a solution for a query works also for similar queries.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

When focusing on the `nice` program, although MEMSIGHT is faster at executing queries in the partition [0.01, 0.1) seconds, it is significantly slower on queries falling in the partition [0.001, 0.01) seconds. As for `strip-new`, if we sum up the cumulative times across different bins, we get a possible explanation for the 36% slowdown reported in Table III. When considering the `nm-new` program, the chart may suggest an overall positive performance impact of MEMSIGHT when running this program under KLEE which however was not observed in Table III, where the measured speedup was negligible. We believe that optimizations from the caching layer in this case may have affected the number and type of queries submitted to the solver during that experiment when using MEMSIGHT instead of standard KLEE. On other benchmarks (such as for the `od` program) where the slowdown is negligible even if there are symbolic accesses and the cumulative query time is significant, the impact of MEMSIGHT was limited since most of the query time is spent in fast queries where our solution appears to be slower than standard KLEE.

Discussion. A first natural question that may arise at this point is why MEMSIGHT appears to perform worse on fast queries when compared to standard KLEE. The answer to this question is related to our way of integrating MEMSIGHT into KLEE. As mentioned in Section 4, MEMSIGHT operates in KLEE by rewriting queries inside the backend solver. This additional pass can add an overhead that is not amortized when the original queries generated by KLEE are solved very quickly by the solver. To avoid this overhead, we should have implemented MEMSIGHT inside the Object State abstraction. However, this would have required us to rewrite also the optimizer layer of KLEE. Although our prototype incurs a performance penalty because of this decision, we believe that an implementation strategy that limits the code changes to the solver backend would more easily allow MEMSIGHT to be integrated in the main KLEE codebase in the near future.

On the contrary, we asked ourselves why MEMSIGHT appears to perform better on queries that required a large solving time under standard KLEE. To answer this question we tried to analyze the type of queries that are submitted to the solver. Since several of them involve arrays whose initial content is constant, we focused our attention on how KLEE handles them. When defining this kind of arrays within a query, KLEE generates a nested `STORE` expression that can be quite deep if the size of the array is considerable. Interestingly, while we were performing this investigation (months after we finished implementing our prototype), the developers of KLEE discovered and patched "a massive performance issue with constant arrays" when using the Z3 solver[**]. Applying their patch results in KLEE using assertions instead of `STORE` expressions to define the initial content of constant arrays. Although the developers noticed this performance problem in Z3 and limited the code changes to the Z3 backend, we extended their patch in order to use assertions for defining constant arrays also within the STP backend. However, we did not observe any appreciable running time improvements on most benchmarks when adopting this patch in STP, obtaining on the other hand slightly worse running times on some of the programs.

We then turned our investigation toward other features of queries that could play a role. In our evaluation, we used symbolic accesses as indicators for performance opportunities that could be exploited by MEMSIGHT. Unfortunately, expressions related to memory pointers embedded in possibly long and complex queries are not easy to detect and analyze statically. At the end, we were not able to find a consistent correlation between the improvements in query time obtained with MEMSIGHT and one single feature in the queries that we considered. We believe it is likely that several different features play a role in the running time, and a solid methodology would be needed in order to pinpoint and correlate them to query running times. A recent work [28] has proposed to use machine learning when deciding which solver to use depending on the features of a given query. Our setting is different, as we wish to compare different queries resulting from handling the same memory access under different memory models, which in turn may leverage different solver facilities. Nonetheless, we believe that a machine learning-assisted technique could help shed some light on this problem, and we hope to see it tackled by the research community in the future.

An alternative investigative approach would have been to walk through the inner workings of the STP solver in the hope of getting insights on why a given query expressed using theory of arrays

[**]https://github.com/klee/klee/issues/836

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

may be more or less efficient compared to when rewritten using the approach of MEMSIGHT, i.e., by using `ITE` expressions in place of `STORE` and `SELECT` expressions. Such an investigation would be far from easy: due to the inherent complexity of modern SMT solvers, the symbolic execution research community has often leveraged them essentially as black boxes. We believe that bridging the gap in knowledge between satisfiability checking theory and its applications in, e.g., software testing research would make a compelling topic for new investigations.

6. RELATED WORK

In Section 2 we outlined the main ideas behind the concept of memory model for symbolic execution and presented the inner workings of the models used by four mainstream symbolic execution engines. However, we left out a number of works that also tackled the problem of handling symbolic pointers. We now briefly review works targeting low-level code that are more relevant for the application contexts targeted by MEMSIGHT.

Memory Models. Concolic executors such as DART [29] and CUTE [30] deal with pointers by exploiting knowledge resulting from the program's concrete execution that is carried alongside the symbolic exploration of a given path. DART uses the concrete value of a pointer in place of its symbolic expression to disambiguate the affected memory location in the presence of a symbolic access. CUTE makes a step further by splitting the values stored in the memory into two different sets: the first set is used to track primitive values, while the second for pointer expressions. This distinction allows CUTE to determine whether a branch condition involves a pointer. CUTE can handle comparisons of equality for pointers, accounting for the different scenarios that might arise because of them in the symbolic exploration.

The memory model of EXE described in the original work [31] supports symbolic reads by emulating pointers as offset references to array objects; concretization is used for multiple pointer dereferences, while symbolic writes are not discussed in detail. The model later went through some extensions and was likely used as the starting point of the one adopted in KLEE [6]. Kapus *et al.* propose in [32] a new segmented model for KLEE to reduce forking by putting objects that may be referenced by the same symbolic pointer into the same segment, represented with a single array.

SAGE [2] takes advantage of concrete values from dynamic test generation to support symbolic pointers, confining them within the memory regions in which the corresponding concrete values fall. The work also discusses the relevance of multiple pointer dereferences and symbolic write operations in software testing.

MAYHEM [16] introduces the partial memory modeling abstraction and proposes a number of clever optimizations such as range refinements with *value-set analysis* [33] and fine-grained query caching to reduce the burden on the SMT solver when assessing range sizes. ANGR implements a partial memory models that closely resembles the one proposed in MAYHEM.

Our MEMSIGHT proposal shares several analogies with the segment-offset-plane model proposed by Trtík *et al.* [34], which stores data in separate planes based on their type. Each plane holds a list of write records, and a solver is invoked for each read operation to check whether a stored expression collides with the given (typed) symbolic address. We believe our approach is more general as it is not affected by the type safety characteristics of a language, it provides support for state merging that is compelling for scalability, and it explicitly accounts for uninitialized memory.

Concretization Strategies. As we have seen in this article, several symbolic executors resort to concretization in the presence of symbolic pointers. The framework presented by David *et al.* [35] describes concretization policies for symbolic values and addresses, shedding light on this research problem and paving the way to a systematic study of concretization strategies and policy tuning. Along these lines, we believe that the delayed concretization technique with uninterpreted functions proposed by Păsăreanu *et al.* [36] to handle non-linear constraints could be value in the setting of pointer concretization strategies too.

Lazy Memory Initialization. A common practice in symbolic execution is to analyze code in isolation. This is often done to avoid the path explosion that would arise when considering the code in the context of the whole program. For instance, a symbolic executor may struggle at reaching code

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

that is buried after a complex input parsing phase. Also, developers of a library may be interested in testing the code library without making any assumption on the client code invoking it.

Nonetheless, code often cannot be executed accurately in isolation. Real-world code may expect a well-defined memory state in order to work properly, i.e., according to the intended application semantics. For instance, a piece of code may behave differently depending on how an internal data structure has been initialized, and a symbolic exploration may end up considering unrealistic memory states, i.e., states that would not be reachable when testing the code in the context of the enclosing program. To allow for a meaningful symbolic exploration even in isolation, different works [37, 38, 39] have investigated how to reconstruct the memory state *lazily*. In particular, several approaches have tackled the problem of lazily reconstructing *heap configurations*, i.e., how to reason over code that inspects dynamically allocated data structures such as linked lists or trees. Most of these techniques rely on the availability of data type information and target object-oriented languages like Java. Although orthogonal to the problems and goals discussed in this article, lazy memory initialization is a very important problem in symbolic execution research, and several of these techniques could be integrated into MEMSIGHT to help it deal with code tested in isolation.

Merging Strategies. In Section 2.2.1 we mentioned merging as a key performance enabler in symbolic executors. Merging comes in two main flavors for deciding when to perform such operation: state merging [3] is well suited to engines that expand different sides of a branch at the same time, while path merging [4] is amenable to implementations where sides are explored one at a time. MEMSIGHT can be used with both approaches. We also note that the tuples used in MEMSIGHT share some analogies with the value summaries created by [40], which also simplifies merging by implicitly representing the possible values that a variable may attain over different paths.

Code Manipulation. The semantics of tuples for memory writes in MEMSIGHT shares analogies with the Static Single Assignment form [41] used in compilers, where each assignment to a variable generates a fresh copy of the variable to hold its most recent value. Similarly, writing to a memory cell in MEMSIGHT generates a new tuple with a higher timestamp, effectively creating a fresh new version of that memory cell. In both cases, read operations always fetch the latest written value.

7. CONCLUSIONS

We believe that the key concept developed in this article of generalizing a symbolic memory model so that it maps *symbolic address expressions*—rather than just concrete addresses—to value expressions, can lead to further interesting developments.

The refinements introduced in Algorithm 2 and the optimizations applied to our prototype implementation can significantly affect the performance of the basic version of the approach. Nonetheless, the optimization design space to explore is large, leaving significant room for improvement. As a first observation, static analysis techniques such as value-set analysis can be used to refine ranges as in MAYHEM [16] and ease constraint solving. Also, the expressions returned by memory load operations could be amenable to simplification, as expressions from recent symbolic writes may together supersede other expressions stored earlier in the execution. Similarly, the paged interval tree may periodically be rebuilt—or modified in a lazy fashion—to prune "outdated" values. Investigating the benefits of delayed pointer concretization in symbolic execution and possible strategies for it remains an interesting open question.

When analyzing the current embodiment of MEMSIGHT from a critical perspective, a few observations are worth to be raised. MEMSIGHT provides a novel approach for efficiently handling symbolic writes, as our memory model does not need to update the mapping for each address possibly affected by such an operation. Nonetheless, its design comes with a few notable drawbacks. Handling symbolic pointers in read operations can still lead MEMSIGHT, similarly to other memory models, to generate lengthy ITE expressions that may become hard to parse for an engine (e.g., when applying rewriting rules) or complex to reason on for an SMT solver. Moreover, keeping additional metadata (e.g., timestamps) to reason over the memory may increase the burden on the symbolic engine, adding spatial and temporal overheads to the exploration. Nonetheless, our experiments with ANGR show that this overhead could be worth the price whenever a partial memory model may

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

perform concretizations that affect the exploration by missing valuable program states. We believe that MEMSIGHT may be able to sustain explorations that cannot be handled by alternative solutions currently available in ANGR. This makes MEMSIGHT appealing in contexts, such as vulnerability exploitation, where symbolic execution may be ineffective due to improper pointer concretizations.

On the other hand, the experiments with KLEE show that when MEMSIGHT is deployed in the context of a framework that treats the memory as a set of independent memory objects, MEMSIGHT may provide a performance level that is comparable to that achieved by a memory model relying on the theory of arrays. Although understanding why MEMSIGHT makes KLEE faster or slower during the exploration of a program is far from being a trivial task, we believe that our discussion of MEMSIGHT may still provide valuable insights. We think that investigating and devising automatic methodologies to pinpoint which factors are impacting the processing time of related queries could bring significant value to the validation of experiments like the ones discussed in Section 5.

In this article, we have defined the memory model as the component supporting a symbolic executor in two aspects: memory object reasoning and memory layout reasoning. However, after implementing MEMSIGHT inside two symbolic execution frameworks, we realized that the design of a memory model should take into account also other aspects. For instance, during our experiments, we identified that two C library functions, `memcpy` and `memset`, constitute a severe challenge. Reading or writing large memory regions can require a non-negligible amount of resources when a symbolic pointer is involved in one of these two operations. The design of memory models should consider these primitives, as well as other memory heavy-duty operations commonly offered by standard programming libraries, in order to improve performance and scalability.

Our experiments seem to suggest that symbolic pointers might not be very common in real-world programs. However, several factors may play a role in the nature of pointers that show up inside a program. For instance, most symbolic executors concretize symbolic expressions that represent object *sizes*. Several memory-related operations often involve a task whose work depends on the size of the object that they are manipulating. Since treating these sizes as symbolic can easily make the exploration unsustainable, symbolic frameworks choose to concretize them. However, symbolic sizes can be extremely common in practice and could easily lead a program to generate symbolic pointers. For instance, primitives offered by the C library for dynamic memory allocation such as `malloc`, `calloc`, or `realloc`, carry out tasks that crucially depend on the allocation size received as argument. In particular, several different memory layouts could be generated by considering different choices when concretizing allocation sizes. We think that the design of a memory model should take into account this problem, possibly providing support for keeping sizes symbolic, without forcing an executor toward their concretization.

REFERENCES

1. Baldoni R, Coppa E, D'Elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. *ACM Computing Surveys* 2018; **51**(3).
2. Elkarablieh B, Godefroid P, Levin MY. Precise pointer reasoning for dynamic test generation. Proceedings of the 18th Int. Symp. on Software Testing and Analysis (ISSTA '09), 2009; 129–140, doi:10.1145/1572272.1572288.
3. Kuznetsov V, Kinder J, Bucur S, Candea G. Efficient state merging in symbolic execution. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2012, ACM, 2012; 193–204, doi:10.1145/2254064.2254088.
4. Avgerinos T, Rebert A, Cha SK, Brumley D. Enhancing symbolic execution with veritesting. Proceedings of the 36th Int. Conference on Software Engineering (ICSE '14), 2014; 1083–1094, doi:10.1145/2568225.2568293.
5. Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Krügel C, *et al.*. SOK: (state of) the art of war: Offensive techniques in binary analysis. Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16), 2016; 138–157, doi:10.1109/SP.2016.17.
6. Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. 8th USENIX Conf. on Operating Systems Design and Implem. (OSDI '08), 2008; 209–224.
7. Coppa E, D'Elia DC, Demetrescu C. Rethinking pointer reasoning in symbolic execution. Proc. of the 32nd IEEE/ACM Int. Conf. on Automated Software Engineering, 2017; 613–618, doi:10.1109/ASE.2017.8115671.
8. King JC. Symbolic execution and program testing. *Comm. of ACM* 1976; :385–394doi:10.1145/360248.360252.
9. SMT-LIB I. The satisfiability modulo theories library 2018. `http://smtlib.cs.uiowa.edu/`.
10. de Moura L, Bjrner N. Generalized, efficient array decision procedures. Proceedings of the 2009 Formal Methods in Computer-Aided Design (FMCAD '09), 2009; 45–52, doi:10.1109/FMCAD.2009.5351142.
11. David R, Bardin S, Ta TD, Mounier L, Feist J, Potet M, Marion J. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. 2016; 653–656, doi:10.1109/SANER.2016.43.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

12. Martignoni L, McCamant S, Poosankam P, Song D, Maniatis P. Path-exploration lifting: Hi-fi tests for lo-fi emulators. Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII), 2012; 337–348, doi:10.1145/2150976.2151012.
13. Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), 2007; 89–100, doi:10.1145/1250734.1250746.
14. De Moura L, Bjørner N. Z3: An efficient smt solver. Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08/ETAPS '08), 2008; 337–340, doi:10.1007/978-3-540-78800-3_24.
15. Shellphish. Cyber Grand Shellphish 2017. `phrack.org/papers/cyber_grand_shellphish.html`.
16. Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing Mayhem on binary code. Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12), 2012, doi:10.1109/SP.2012.31.
17. Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07), 2007; 519–531.
18. Chipounov V, Kuznetsov V, Candea G. S2e: A platform for in-vivo multi-path analysis of software systems. Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI), 2011; 265–278, doi:10.1145/1950365.1950396.
19. Sasnauskas R, Landsiedel O, Alizai MH, Weise C, Kowalewski S, Wehrle K. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. Proc. of the 9th ACM/IEEE Int. Conference on Information Processing in Sensor Networks (IPSN '10), 2010; 186–196, doi:10.1145/1791212.1791235.
20. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '05), 2005; 190–200, doi:10.1145/1065010.1065034.
21. Djoudi A, Bardin S. Binsec: Binary code analysis with low-level regions. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '15), 2015; 212–217.
22. David R. Formal approaches for automatic deobfuscation and reverse-engineering of protected codes. PhD Thesis. Universit de Lorraine, 2017. `http://www.robindavid.fr/publications/thesis_RD.pdf`.
23. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*. 3rd edn., The MIT Press, 2009.
24. Perry DM, Mattavelli A, Zhang X, Cadar C. Accelerating array constraints in symbolic execution. Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17), 2017; 68–78, doi:10.1145/3092703.3092728.
25. Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS '16), 2016.
26. Shoshitaishvili Y, Wang R, Hauser C, Kruegel C, Vigna G. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15), 2015, doi:10.14722/ndss.2015.23294.
27. Baldoni R, Coppa E, D'Elia DC, Demetrescu C. Assisting malware analysis with symbolic execution: A case study. Proc. of the First Int. Conference on Cyber Security Cryptography and Machine Learning (CSCML '17), 2017.
28. Dustmann OS, Rath F, Martin P, Wehrle K. Choosing the best solver for your query. KLEE Workshop 2018, 2018.
29. Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2005; 213–223, doi:10.1145/1065010.1065036.
30. Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for c. Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '13), 2005; 263–272, doi:10.1145/1081706.1081750.
31. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: Automatically generating inputs of death. Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS '06), 2006, doi:10.1145/1180405.1180445.
32. Kapus T, Cadar C. A segmented memory model for symbolic execution. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, 2019; 774–784, doi:10.1145/3338906.3338936.
33. Balakrishnan G, Reps T. Analyzing memory accesses in x86 executables. Proceedings of the 13th International Conference on Compiler Construction (CC '04), 2004; 5–23.
34. Trtík M, Strejček J. Symbolic memory with pointers. Proceedings of 12th International Symposium on Automated Technology for Verification and Analysis (ATVA '14), 2014; 380–395, doi:10.1007/978-3-319-11936-6_27.
35. David R, Bardin S, Feist J, Mounier L, Potet ML, Ta TD, Marion JY. Specification of concretization and symbolization policies in symbolic execution. Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16), 2016, doi:10.1145/2931037.2931048.
36. Păsăreanu CS, Rungta N, Visser W. Symbolic execution with mixed concrete-symbolic solving. Proceedings of the 2011 Int. Symposium on Software Testing and Analysis (ISSTA '11), 2011, doi:10.1145/2001420.2001425.
37. Khurshid S, Păsăreanu CS, Visser W. Generalized Symbolic Execution for Model Checking and Testing. Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03), 2003; 553–568, doi:10.1007/3-540-36577-x_40.
38. Engler D, Dunbar D. Under-constrained execution: Making automatic code destruction easy and scalable. Proc. of the 2007 Int. Symp. on Software Testing and Analysis (ISSTA '07), 2007; 1–4, doi:10.1145/1273463.1273464.
39. Geldenhuys J, Aguirre N, Frias MF, Visser W. Bounded lazy initialization. Proceedings of the 5th International NASA Formal Methods Symposium (NFM '13), 2013; 229–243, doi:10.1007/978-3-642-38088-4_16.
40. Sen K, Necula G, Gong L, Choi W. Multise: Multi-path symbolic execution using value summaries. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 2015; 842–853, doi:10.1145/2786805.2786830.
41. Alpern B, Wegman MN, Zadeck FK. Detecting equality of values in programs. *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, 1988.

Copyright © 2019 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls