Rethinking Pointer Reasoning in Symbolic Execution

Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu
Department of Computer, Control, and Management Engineering
Sapienza University of Rome, Italy
{coppa, delia, demetres}@dis.uniroma1.it

***Abstract*—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 concretize symbolic addresses that span memory intervals larger than some threshold. Unfortunately, this could result in missing interesting execution states, e.g., where a bug arises.**

In this paper we introduce MemSight, a new approach to symbolic memory that reduces the need for concretization, hence offering the opportunity for broader state explorations and more precise pointer reasoning. Rather than mapping address instances to data as previous tools 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. A preliminary experimental investigation on prominent benchmarks from the DARPA Cyber Grand Challenge shows that MemSight enables the exploration of states unreachable by previous techniques.

> *Blessed are the forgetful, for they get the better even of their blunders.*
>
> *Friedrich Nietzsche*

I. 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, or when a pointer to be dereferenced is represented by a symbolic expression. We base our discussion on the simple running example reported in Figure 1.

```
void bomb(char* a, char i, char j) {
    char boom;
    a[i] = 23;
    if (a[j] == 23) boom = 0;
    else            boom = 1;
    assert(!boom);
}
```

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

The function takes as inputs an array `a` and two indexes `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., that do not trigger the `assert` statement. Previous research and state-of-the-art tools typically model memory as a mapping between concrete addresses and expressions over concrete and symbolic values. In this setting, different scenarios become possible for handling a symbolic address when referencing memory.

A symbolic executor can *concretize* an address by using one valid model for the symbolic expression under the current path constraints. This strategy naturally arises for instance in dynamic test generation, in which values from a concrete execution are maintained at the same time. Previous research has however pointed out that concretization can fail to exercise program branches and paths [2]. On the other hand, treating the memory as *fully symbolic* would allow an executor to reason about all possible addresses either by forking the execution to account for each concrete address matching an expression, or by capturing the uncertainty on the address through nested ite (i.e., if-then-else) expressions over its possible values.

Unfortunately, fully symbolic memory as described above hardly scales in practice. Hence, state-of-the-art executors often trade performance for soundness by implementing a *partial* memory model in which writes are always concretized, while reads are modeled as in fully symbolic memory only in the face of a manageable number (e.g., up to 1024 in [3]) of possible address values, and concretized otherwise.

To simplify the discussion of our example, let us assume that the involved memory is initially zero-filled, so to avoid bomb defusing from pre-existing storage. Full concretization of both `&a[i]` and `&a[j]` would likely result in an assertion failure, unless the same value is used for `i` and `j`. A partial memory model would concretize the symbolic read too, 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 would instead reveal the property that all inputs in which `i==j` holds defuse the bomb.

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research - New Ideas

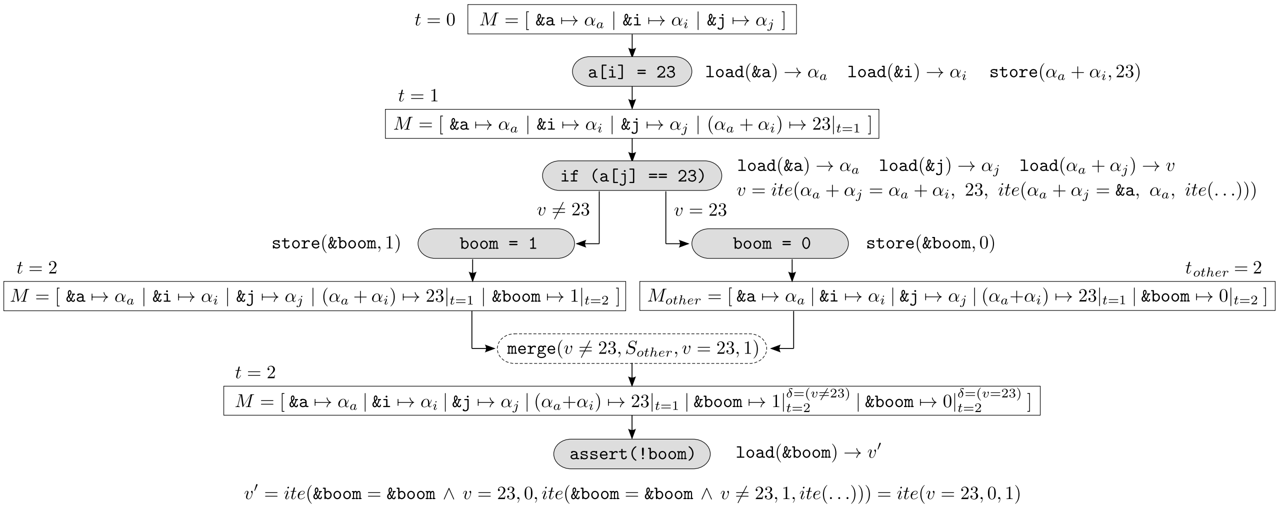

$t = 0$
$M = [\ \texttt{\&a} \mapsto \alpha_a \mid \texttt{\&i} \mapsto \alpha_i \mid \texttt{\&j} \mapsto \alpha_j\ ]$
a[i] = 23
$\texttt{load(\&a)} \to \alpha_a$ $\texttt{load(\&i)} \to \alpha_i$ $\texttt{store}(\alpha_a + \alpha_i, 23)$
$t = 1$
$M = [\ \texttt{\&a} \mapsto \alpha_a \mid \texttt{\&i} \mapsto \alpha_i \mid \texttt{\&j} \mapsto \alpha_j \mid (\alpha_a + \alpha_i) \mapsto 23|_{t=1}\ ]$
if (a[j] == 23)
$\texttt{load(\&a)} \to \alpha_a$ $\texttt{load(\&j)} \to \alpha_j$ $\texttt{load}(\alpha_a + \alpha_j) \to v$
$v = ite(\alpha_a + \alpha_j = \alpha_a + \alpha_i,\ 23,\ ite(\alpha_a + \alpha_j = \texttt{\&a},\ \alpha_a,\ ite(\ldots)))$
$v \neq 23$
$v = 23$
$\texttt{store(\&boom}, 1)$
boom = 1
boom = 0
$\texttt{store(\&boom}, 0)$
$t = 2$
$M = [\ \texttt{\&a} \mapsto \alpha_a \mid \texttt{\&i} \mapsto \alpha_i \mid \texttt{\&j} \mapsto \alpha_j \mid (\alpha_a + \alpha_i) \mapsto 23|_{t=1} \mid \texttt{\&boom} \mapsto 1|_{t=2}\ ]$
$t_{other} = 2$
$M_{other} = [\ \texttt{\&a} \mapsto \alpha_a \mid \texttt{\&i} \mapsto \alpha_i \mid \texttt{\&j} \mapsto \alpha_j \mid (\alpha_a + \alpha_i) \mapsto 23|_{t=1} \mid \texttt{\&boom} \mapsto 0|_{t=2}\ ]$
$\texttt{merge}(v \neq 23, S_{other}, v = 23, 1)$
$t = 2$
$M = [\ \texttt{\&a} \mapsto \alpha_a \mid \texttt{\&i} \mapsto \alpha_i \mid \texttt{\&j} \mapsto \alpha_j \mid (\alpha_a + \alpha_i) \mapsto 23|_{t=1} \mid \texttt{\&boom} \mapsto 1|^{\delta=(v\neq 23)}_{t=2} \mid \texttt{\&boom} \mapsto 0|^{\delta=(v=23)}_{t=2}\ ]$
assert(!boom)
$\texttt{load(\&boom)} \to v'$
$v' = ite(\texttt{\&boom} = \texttt{\&boom} \wedge v = 23, 0, ite(\texttt{\&boom} = \texttt{\&boom} \wedge v \neq 23, 1, ite(\ldots))) = ite(v = 23, 0, 1)$

Fig. 2. 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 $\alpha_i = \alpha_j$.

Contributions. In this work we discuss a different perspective in the design of symbolic pointer reasoning: we show how to compactly associate values with symbolic address expressions rather than concrete addresses, and investigate efficient implementations of a fully symbolic memory using a *paged interval tree*. Our approach, which we call MEMSIGHT, natively accounts for state merging [4] – a mainstream performance enabler in modern executors – and has been integrated in the ANGR [5] framework. Preliminary results show that MEMSIGHT allows for broader state exploration on prominent benchmarks, revealing behaviors that previous techniques would miss or use too much resources to identify.

II. 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. Data is stored in a memory object accessible through *load* and *store* operations over expressions describing addresses. 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 a merge of the respective memories as well.

A. Base Version

We model 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 merge of memories, we account for a predicate δ reflecting specific conditions under which the tuple is valid: δ is typically computed by the executor in terms of diverging path constraints between the states to be merged.

The base version of our symbolic memory data structure 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 indexes `i` and `j` with symbolic values α_i and α_j, respectively.

The program's effects on the symbolic state are illustrated in Figure 2. 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 $e \mapsto v|^{\delta=\cdots}_{t=\ldots}$, omitting t if $t = 0$, and δ if $\delta = true$.

Algorithm 1 MEMSIGHT – base version

```
$M$  := symbolic memory (initially empty)
$t$  := timer (initially 0)

function STORE($e, v$):
    $t \leftarrow t + 1$
    $M \leftarrow M \cup \{(e, v, t, true)\}$

function LOAD(e):
    $v \leftarrow 0$
    for $x \in M$ by ascending timestamp do
        $v \leftarrow ite(e = x.e \wedge x.\delta,\ x.v,\ v)$
    return $v$

function MERGE($\delta, S_{other}, \delta_{other}, t_a$):
    for $\{x \in M \mid x.t > t_a\}$ do
        $M \leftarrow M|_{x.\delta \leftarrow x.\delta \wedge \delta}$
    for $\{x \in S_{other}.M \mid x.t > t_a\}$ do
        $x.\delta \leftarrow x.\delta \wedge \delta_{other}$
        $M \leftarrow M \cup \{x\}$
    $t \leftarrow max(t, S_{other}.t)$
```

1) Memory Loading and Storing: To perform `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 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 off a new state $S_{other} = \{M_{other}, t_{other}\}$ for the "then" branch, where M_{other} is a clone of M and $t_{other} = t$. This branch is taken iff $v = 23$.

2) 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: $\delta = (v \neq 23)$ is the path condition of the "else" branch that kept on working on M; S_{other} is the state resulting from the "then" branch, while $\delta_{other} = (v = 23)$ is its path condition; finally, $t_a = 1$ is the timestamp of the least common ancestor 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 "else" branch condition δ (lines 2–3), and then adds to M all tuples added to M_{other} since t_a, guarding them with the "then" branch condition δ_{other} (lines 4–6). In our example, this results in tuples $\texttt{\&boom} \mapsto 1|^{\delta=(v\neq23)}_{t=2}$ and $\texttt{\&boom} \mapsto 0|^{\delta=(v=23)}_{t=2}$ present in M after merging the states (Figure 2).

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.

B. Refinements

In its initial naive formulation, the proposed scheme suffers from a few generality and performance issues. We discuss a number of refinements that lead to Algorithm 2.

1) 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. We note that it is common for a symbolic address to be constrained within a certain interval [3]. 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 just the tuples whose ranges intersect with the minimum and maximum values of e (lines 2–3, 8).

2) Memory Cleanup: Algorithm 1 naively adds one tuple at each `store`. 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.

3) 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 zero, which limits the precision of the analysis. The `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. More precisely, for any two `load`$(e) = v$ and `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

Algorithm 2 MEMSIGHT – improved version

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

 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:     $M \leftarrow M \cup \{(a, b, e, v, t, true)\}$

 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:     $v \leftarrow 0$
 8:     for $x \in range(M, a, b)$ by ascending timestamp do
 9:         $v \leftarrow ite(e = x.e \wedge x.\delta,\, x.v,\, v)$
10:     return $v$

 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:     for $\{x \in S_{other}.M \mid x.t > t_a \vee x.t < \bar{t}_a\}$ do
 5:         $x.\delta \leftarrow x.\delta \wedge \delta_{other}$
 6:         $M \leftarrow M \cup \{x\}$
 7:     $t \leftarrow max(t, S_{other}.t)$
 8:     $\bar{t} \leftarrow min(\bar{t}, S_{other}.\bar{t})$
```

how constraint solvers deal with uninterpreted functions.

4) Multi-byte Load and Store: The solutions presented in this section work with 1-byte memory objects. Multi-byte operations can be supported by issuing separate `store` and `load` operations for individual bytes and combining the results. For instance, a `load`$(e,$`sizeof(int)`$)$ can be obtained by concatenating the values produced by `load`(e), `load`$(e+1)$, `load`$(e+2)$, and `load`$(e+3)$. This strategy is adopted in several symbolic executors (e.g., KLEE[1][6]).

C. Discussion

Previous work [2] hinted that certain bugs can only be revealed when writes are modeled symbolically. However, classic approaches may not scale. For instance, [7] remarks 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 require an explicit enumeration – a possibly expensive task for a solver [3] – of valid addresses to either fork the state or build *ite* expressions (Section I). As we discuss next, range intersection operations can be offloaded to efficient data structures. Our example also suggests that reasoning over expressions allows for an easier input class characterization.

III. Implementation

In this section we describe a prototype implementation of MemSight as a plugin of angr. We start by presenting angr, and then describe the techniques implemented in our prototype while seeking for efficiency.

A. angr

angr [5] is an open-source framework for binary analysis. It provides a powerful platform-agnostic symbolic execution engine that reasons on VEX bytecode and relies on Z3 as SMT solver. angr recently participated in the DARPA Cyber Grand Challenge and has received considerable attention from many reverse engineering and security practitioners.

angr adopts a partial memory model in which *ite* expressions are constructed for a symbolic read if the spanned range is not too large. Z3 is queried for the maximum and minimum values that the address can assume, and if they differ at most by 1024, angr will ask Z3 to enumerate all the solutions and construct an *ite* accordingly, otherwise the address will be concretized. Optionally, write addresses can be treated as symbolic too, with a default threshold of 128 for range size.

In order to relieve the solver from the burden of repeated queries and improve efficiency, a number of optimizations are implemented in the claripy constraint-solving wrapper. To improve scalability, angr implements an extended *veritesting* merging strategy [4] that analyzes the control flow graph to determine at which places is profitable to condense the effects of separate chunks of code using *ite* constraints.

B. MemSight *Prototype*

We devise MemSight as a plugin[2] implementing the memory abstraction required by the simuvex 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.

The first practical challenge we have to overcome is supporting `range` operations from Algorithm 2 efficiently. One possibility is to maintain an *interval tree* to allow for efficient retrieval of all stored intervals that overlap with a given one. However, we should keep in mind that when a branch is encountered, an executor typically clones the state along with the associated memory. To allow better 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 indexes 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.

Another 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 is implemented as a paged hashmap with copy-on-write cloning for pages, similarly as in angr's default memory implementation.

IV. Evaluation

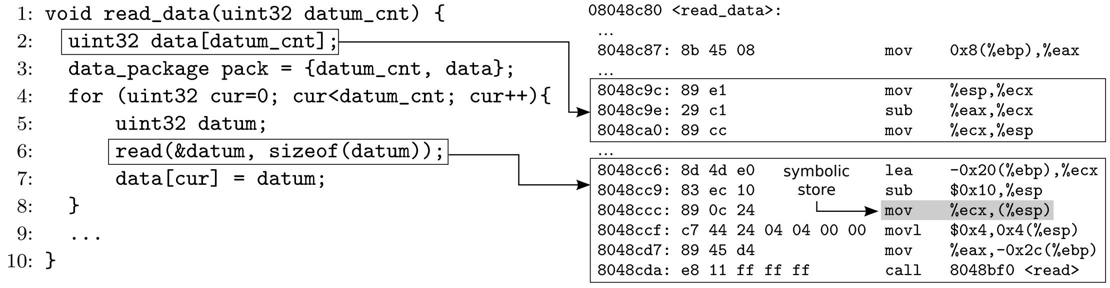
In this section we report on a preliminary investigation of the practical impact of MemSight compared to previous memory representations for symbolic execution.

A. Experimental Setup and Methodology

Our experiments are based on benchmarks from the Cromulence (CROMU) DARPA performer group of the Cyber Grand Challenge (CGC). 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) angr-conc, which concretizes all accessed symbolic addresses, 2) angr-part (the default of angr), which concretizes all read addresses that span ranges larger than 1024 and all written addresses, and 3) angr-full,

[1] https://github.com/klee/klee/blob/master/docs/overview.

[2] 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.



```
 1: void read_data(uint32 datum_cnt) {
 2:   uint32 data[datum_cnt];
 3:   data_package pack = {datum_cnt, data};
 4:   for (uint32 cur=0; cur<datum_cnt; cur++){
 5:       uint32 datum;
 6:       read(&datum, sizeof(datum));
 7:       data[cur] = datum;
 8:   }
 9:   ...
10: }
```

```
08048c80 <read_data>:
 ...
 8048c87: 8b 45 08                 mov    0x8(%ebp),%eax
 ...
 8048c9c: 89 e1                    mov    %esp,%ecx
 8048c9e: 29 c1                    sub    %eax,%ecx
 8048ca0: 89 cc                    mov    %ecx,%esp
 ...
 8048cc6: 8d 4d e0                 lea    -0x20(%ebp),%ecx
 8048cc9: 83 ec 10                 sub    $0x10,%esp
 8048ccc: 89 0c 24                 mov    %ecx,(%esp)
 8048ccf: c7 44 24 04 04 00 00 00  movl   $0x4,0x4(%esp)
 8048cd7: 89 45 d4                 mov    %eax,-0x2c(%ebp)
 8048cda: e8 11 ff ff ff           call   8048bf0 <read>
```

Fig. 3. Example of symbolic store excerpted from the CROMU_00006 Cyber Grand Challenge benchmark.

which performs no address concretizations by raising the read and write thresholds to an infinite value.

We symbolically execute MEMSIGHT, ANGR-CONC, ANGR-PART, and ANGR-FULL 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.

B. 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 included in the DARPA CGC suite that produces random numbers and generates charts for numeric data, including bar charts and sparklines. The benchmark dereferences pointers that span very large portions of the address space. This arises for instance in function `read_data` shown in Figure 3, 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` causes a store to the symbolic `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 ANGR-PART, forcing the symbolic execution to reason on a buffer of fixed size[3]. Since the symbolic range for `esp` is very large, ANGR-FULL 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, 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 ANGR-PART.

C. Experiments

The first question we address is: how broad are the symbolic address dereferences performed by the CROMU benchmarks? To this aim, we measure the range of symbolic loads and stores throughout the analysis. This is a structural property of the considered benchmarks independent of the chosen memory model. The left half of Table I reports the total number of memory accesses to a symbolic address with a range size larger than 1 (# CONCR) and the maximum size of the ranges of symbolic addresses accessed by load and store operations throughout the execution. Notice that for some benchmarks the ranges are much larger than the thresholds one can afford to use in practice in partial memory models.

Our second question is: to what extent does concretization restrict state explorations? The right part of Table I compares the number of distinct control flow paths explored by the memory models we considered. To allow direct comparison, the snapshot of the number of paths is taken at the same exploration depth K for the same benchmark in all memory versions. We first observe that full concretization (ANGR-CONC) may restrict the number of explorable paths, confirming the findings reported in [2], [3]. We also note that for some benchmarks that exhibit large ranges for symbolic addresses, a partial memory model (ANGR-PART) does not capture all explorable paths. Explicit fully symbolic memory (ANGR-FULL) fails to complete due to excessive resource requirements. Full exploration is instead supported by MEMSIGHT, which can visit a larger portion of the execution state space.

V. RELATED WORK

A number of projects have addressed the problem of modeling symbolic pointers. The memory model of EXE [8] allows 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. KLEE [6] implements a similar strategy, but clones the execution state when a pointer can refer to multiple objects, constraining the pointer to be within a single object in a clone.

SAGE [2] takes advantage of concrete values from dynamic test generation to support symbolic pointers, confining them

[3] 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.

TABLE I
PRELIMINARY EXPERIMENTS ON THE CROMU CYBER GRAND CHALLENGE BENCHMARKS.

BENCHMARK	# CONCR	MAX RANGE SIZE		# PATHS AT ROUND K				
		LOAD	STORE	K	ANGR-CONC	ANGR-PART	ANGR-FULL	MEMSIGHT
CROMU_00001	230	0	9	481	1845	1845	2466	2466
CROMU_00006	6	24	262128	316	9	31	failed	**34**
CROMU_00009	638	8	9	1534	1131	1131	1715	1715
CROMU_00014	1800	0	9	1512	2251	2251	2315	2315
CROMU_00018	1696	400	770048	924	80	98	failed	**433**
CROMU_00024	1902	32	32	327	1563	1563	1980	1980
CROMU_00027	771	92	3888	3522	2142	2142	2192	2192
CROMU_00031	126	2295	56	899	299	1413	1413	1413
CROMU_00032	193	576	8192	642	3	3	51	51
CROMU_00033	1010	1020	1020	364	508	508	539	539

within the memory regions in which the corresponding concrete values fall. The work also discusses the relevance of multiple pointer dereferences and symbolic writes in testing.

MAYHEM [3] introduces partial memory modeling, and proposes a number of clever optimizations such as *value-set analysis* [9] and fine-grained query caching to reduce the burden on the SMT solver when assessing range sizes.

Our work shares several analogies with the segment-offset-plane model proposed in [10], 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 of a language, it provides support for state merging that is compelling for scalability, and it explicitly accounts for uninitialized memory.

The framework presented in [11] to describe concretization policies for symbolic values and addresses sheds light on an interesting research problem, paving the way to a systematic study of concretization strategies and policy tuning. We also believe that pointer concretization strategies might benefit from the delayed concretization technique with uninterpreted functions proposed in [12] to handle non-linear constraints.

VI. CONCLUSION

We believe that the key concept of generalizing a symbolic memory 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 design space to explore in optimization 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 [3] and ease constraint solving. Also, the expressions returned by `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.

An executor might also decide to trade performance for soundness at a later stage by concretizing certain symbolic address expressions, or limiting the ranges they span using speculative heuristics. Investigating the benefits of delayed pointer concretization in symbolic execution and possible strategies for it remains an interesting open question.

ACKNOWLEDGMENTS

This work is partially supported by a grant of the Italian Presidency of Ministry Council and by CINI Cybersecurity National Laboratory within the project FilieraSicura funded by CISCO Systems Inc. and Leonardo SpA.

REFERENCES

[1] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *CoRR*, vol. abs/1610.00502. [Online]. Available: http://arxiv.org/abs/1610.00502

[2] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *Proc. of ISSTA 2009*. ACM, 2009. [Online]. Available: https://doi.org/10.1145/1572272.1572288

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *Proc. of SP 2012*. IEEE Computer Society, 2012. [Online]. Available: https://doi.org/10.1109/SP.2012.31

[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. of ICSE 2014*. ACM, 2014. [Online]. Available: https://doi.org/10.1145/2568225.2568293

[5] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. of SP 2016*. IEEE Computer Society, 2016. [Online]. Available: https://doi.org/10.1109/SP.2016.17

[6] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of OSDI 2008*. USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

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

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *CCS 2006*. ACM, 2006. [Online]. Available: https://doi.org/10.1145/1180405.1180445

[9] E. Duesterwald, Ed., *Analyzing Memory Accesses in x86 Executables*, ser. CC 2004. Springer Berlin Heidelberg, 2004. [Online]. Available: https://doi.org/10.1007/978-3-540-24723-4_2

[10] M. Trtík and J. Strejček, *Symbolic Memory with Pointers*, ser. ATVA 2014. Cham: Springer International Publishing, 2014, pp. 380–395. [Online]. Available: https://doi.org/10.1007/978-3-319-11936-6_27

[11] R. David, S. Bardin, J. Feist, L. Mounier, M.-L. Potet, T. D. Ta, and J.-Y. Marion, "Specification of concretization and symbolization policies in symbolic execution," in *Proc. of ISSTA 2016*. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2931037.2931048

[12] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *Proc. of ISSTA 2011*. ACM, 2011. [Online]. Available: https://doi.org/10.1145/2001420.2001425