Handling Memory-Intensive Operations in Symbolic Execution

Luca Borzacchiello
borzacchiello@diag.uniroma1.it
Sapienza University of Rome
Rome, Italy

Emilio Coppa
coppa@diag.uniroma1.it
Sapienza University of Rome
Rome, Italy

Camil Demetrescu
demetres@diag.uniroma1.it
Sapienza University of Rome
Rome, Italy

ABSTRACT

Symbolic execution is a popular software testing technique that can help developers identify complex bugs in real-world applications. Unfortunately, symbolic execution may struggle at analyzing programs containing memory-intensive operations, such as `memcpy` and `memset`, whenever these operations are carried out over memory blocks whose size or address is symbolic, i.e., input-dependent.

In this paper, we devise MInt, a memory model for symbolic execution that can support reasoning over such operations. The key new idea behind our proposal is to make the memory model *aware* of these memory-intensive operations, deferring any symbolic reasoning on their effects to the time where the program actually manipulates the symbolic data affected by these operations. We show that a preliminary implementation of MInt based on the symbolic framework angr can effectively analyze applications taken from the DARPA Cyber Grand Challenge.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; **Software testing and debugging**.

KEYWORDS

symbolic execution, software testing

ACM Reference Format:
Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2022. Handling Memory-Intensive Operations in Symbolic Execution. In *15th Innovations in Software Engineering Conference (ISEC 2022), February 24–26, 2022, Gandhinagar, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3511430.3511453

1 INTRODUCTION

Symbolic execution (SE) [1, 3, 4, 6, 11, 23, 24] is a software testing technique that evaluates the behavior of a program on *symbolic*, rather than *concrete*, inputs. A symbolic input can assume any value that is admissible for its data type, e.g., a symbolic `uint8_t` can evaluate to any value in the range `[0, 255]`. To model the computation over symbolic data, SE builds symbolic formulas. Whenever a program may take a different execution direction due to a branch condition over inputs, SE submits symbolic queries to a constraint solver [15] in order to evaluate whether both branch directions are admissible for some values of the input data. If this is the case, SE forks the current execution path, adding in each subpath the constraint derived from the taken branch direction. We refer the reader to a recent survey [4] on symbolic execution for a more detailed discussion of this program analysis technique.

Several challenges make SE hard to apply in presence of real-world code [4, 11]. In this paper, we tackle one fundamental but hard-to-solve aspect: symbolic reasoning over the memory state of a program, which is often referred in literature with the term *symbolic memory model* [5, 20]. In particular, the main challenge in a memory model for SE is how to handle *symbolic memory accesses*, i.e., memory accesses whose address depends on the value of input data. For instance, a program may update the content of an element within an array using an index whose value is input-dependent: SE may struggle at reasoning over the effects of this operation.

State-of-the-art symbolic frameworks adopt different memory models. For instance, KLEE [9] resorts to the support theory of arrays [18] from SMT solvers [14, 19] to reason over symbolic accesses within an *object* (such as an array), forking the execution state whenever the memory access may affect different objects. Since determining the size of objects at binary-level could be hard, binary frameworks, such as angr [25], trade accuracy for scalability, considering only a subset of the admissible values for a symbolic address, e.g., they *concretize* the memory address to a single concrete value whenever the address may span too many memory locations.

While several memory models have been proposed in the last decades [10, 20, 28], one challenge that has been not considered thoroughly is how to handle efficiently memory-intensive operations, such as `memset` and `memcpy`. Indeed, although these functions could be regarded just as sequences of `load` and `store` operations, when treated in this way, they may stress the limitations of the underlying memory model, affecting the scalability and effectiveness of SE.

Motivating examples. To better understand the challenges resulting from memory-intensive operations, we consider three motivating examples, excerpted from real-world code.

Figure 1a shows a function `foo` that first performs a `memset` operation over the first `n` bytes of a buffer (`buf`) and then checks a condition over the byte at position `X` of the buffer. If we assume that `n` is symbolic while `X` is a non-negative constant and both of them are smaller than the size of the buffer (i.e., no out-of-bounds access can happen), then an execution may reach one of the two return statements depending on the value of `n`: if `X < n` then the first `return` is executed, otherwise the second `return` is executed. To the best of our knowledge, no memory model in mainstream symbolic

```
1. // n is symbolic
2. // n is smaller than size of buf
3. // X is a const. idx < size of buf
4. int foo(char* buf, size_t n) {
5.   memset(buf, 'a', n);
6.   if (buf[X] == 'a')
7.     return 0; // reachable
8.   else return 1; // reachable
9.}
```

(a)

```
1. // n is symbolic, n < size of buf
2. // N is a constant positive value
3. int bar(char* buf, size_t n) {
4.   memset(buf, 'a', n);
5.   if (n != N) return 0; // reachable
6.   if (buf[N - 1] != 'a')
7.     return 2; // unreachable
8.   return 3; // reachable
9.}
```

(b)

```
1. // n is symbolic
2. // n is smaller than size of src and dst
3. // X is a constant idx < size of dst
4. // buffer at src is zero-initialized
5. int baz(char* dst, char* src, size_t n) {
6.   memcpy(dst, src,  n);
7.   if (dst[X] == 0x0) return 0; // reachable
8.   else return 1; // reachable
9.}
```

(c)

Figure 1: Examples: (a) `memset` with a symbolic size and two reachable `returns`, (b) `memset` with a symbolic size and three `returns` where only two are reachable, and (c) `memcpy` with a symbolic size and two reachable `returns`.

frameworks is aware of `memset` operations. For instance, KLEE would execute symbolically the code of `memset`, possibly yielding path explosion when the buffer size could be large: the standard implementation of `memset` integrates a conditional loop over the buffer size. Other frameworks may detect the `memset` procedure and reason on it using an *API model*, which compactly describes the effects of an API operation. However, even an API model cannot solve the challenge posed by `memset` without specific support from the underlying memory model: for instance, ANGR – for scalability reasons – would evaluate the effects of `memset` considering only a restricted number of values for `n` (e.g., $n \in [0, 48]$). In general, the problem is even worse when considering a pointer `buf` that is symbolic. Hence, depending on the size of the buffer and the value of `X` (e.g., `X=65534` and `buf` of 65535 bytes), several symbolic frameworks may struggle at exploring both `return` statements.

While the first example shows how existing symbolic frameworks may fail to explore reachable states, the second example can instead show one scenario where some symbolic frameworks may explore unreachable code. Function `bar` in Figure 1b has one return statement that is unreachable: at line 7, `n` must be equal to `N` due to the check at line 6 and thus there is no execution state where `buf[N - 1] != 'a'`. However, symbolic frameworks, such as ANGR, which trade accuracy for scalability, may model the effects of `memset` in a *restricted* way, possibly generating execution paths even for unreachable code. In this example, ANGR would only write the first 48 bytes of the buffer during the `memset` operation.

The last example (Figure 1c) considers a function `baz` executing a `memcpy` operation. Similar to the first example, we assume that `n` is symbolic, `X` is constant, and both `n` and `X` are smaller than the size of the buffers. Additionally, we assume that the buffer pointed by `src` has been zero-initialized. As for `memset`, KLEE would symbolically analyze the implementation of `memcpy`, possibly generating path explosion when `n` may be large. Reasoning on `memcpy` using an API model could be still suboptimal: e.g., ANGR limits the number of bytes that are moved by the `mempcy` operation to at most 4096 bytes. In general, several symbolic frameworks may fail at generating states reaching the two return statements.

Our contribution. In this paper, we devise MINT, a new memory model for SE that is aware of two very common memory-intensive operations, `memcpy` and `memset`, respectively. Our proposal is based on MEMSIGHT, a recently proposed memory model that can deal with symbolic accesses but still struggles in the case of memory-intensive operations. Section 2 explains the inner workings of MEMSIGHT, while Section 3 presents our new ideas. In Section 4, we present a preliminary evaluation of MINT when integrated into ANGR. Finally, Section 5 provides insights on future directions for our proposal.

2 BACKGROUND

Several memory models for SE have been proposed over the last years [9, 10, 12, 13, 16, 20, 28]. KLEE [9] uses the theory of arrays [18] to reason over the content of a memory object, forking the state whenever a memory access may affect different objects. Recent refinements [20, 27] reduces the number of forks. To cope with `memcpy` and `memset`, Falke et. al. [17] proposed to extend the theory of arrays, however, their ideas have never been integrated into mainstream SMT solvers. When the theory of arrays is not used, a common approach is the *flat memory model*, which treats each memory region as an array mapping each address to an 8-bit bit-vector: to deal with symbolic accesses, the symbolic executor *concretizes* the value of each pointer, considering one (or a small subset) of the admissible values to favor scalability. When a subset of the values is considered, the symbolic executor may use *if-then-else* expressions to compactly represent the different execution states [12, 25]. This strategy is often adopted by concolic executors, such as SYMCC [22] and FUZZOLIC [7, 8], or binary symbolic frameworks, such as ANGR [25]. More sophisticated approaches, e.g., [13, 16, 21, 28], have been devised but they do not target memory-intensive operations. In this paper, we extend MEMSIGHT [13], a recently proposed memory model.

MemSight. Figure 2 shows the pseudocode of MEMSIGHT using lines with a white background. The memory model exposes two main primitives: STORE to store data into the memory and LOAD to load data from the memory.

Given an address e (which could be either concrete or symbolic), a (symbolic) data object V, the *size* in bytes of V, and a condition δ[1], STORE uses the solver to get the minimum and maximum concrete addresses that are admissible for e, it increments a global timestamp t (initially set to zero), and then for each (symbolic) byte V_k of V, it stores it into either the concrete memory M_c (when e can assume a single value) or into the symbolic memory M_s. Internally, M_c and M_s keep track of tuples containing: an id for the memory in which the tuple is inserted, the minimum (a) and the maximum (b) value for the address e, the address e, the expression V_k, the time t, and the condition δ. M_c is implemented as a hash-table between concrete addresses and tuples, while M_s is implemented as an interval tree that uses $[a, b]$ for indexing the tuples, allowing fast lookups given an address range $[x, y]$.

[1]Required for handling conditional store operations, which are needed. e.g., to support state merging [2]. On standard store operations, δ is $true$.

The LOAD primitive takes as arguments an address *e* (which could be either concrete or symbolic) and the number of bytes (*size*) to fetch from the memory. For each byte, LOAD calls a subroutine _LOAD to retrieve the (symbolic) expression V_k and then concatenates the byte-expressions into a single object V. _LOAD uses the constraint solver to get the minimum (a) and maximum (b) concrete addresses for *e*, it uses the range $[a, b]$ to retrieve a list of tuples P from M_c and M_s using the subroutine SEARCH, which returns a list sorted based on the timestamp. Finally, _LOAD iteratively builds a nested *if-then-else* (*ite*) expression that takes into account whether the data $x.v$ associated to a tuple x from the list may be accessed by the address *e* (considering also the condition $x.\delta$ in case x was generated by a conditional store).

When *max* and *min* queries are optimized (e.g., through query caching), MemSight is relatively efficient at handling symbolic store operations since it only needs to update M_c and M_s. Symbolic load operations may be more expensive to handle as they require building nested *ite* expressions. However, the overall cost will depend on the number of *conflicts* on the address ranges tracked by the memory model: experimentally, the number of *conflicts* is expected to be small [13]. A nice benefit of MemSight is thus that it can scale even when a program is performing a store operation using a symbolic address *e* with a very large range $[a, b]$. Unfortunately, MemSight is not aware of memory-intensive primitives, treating them as a sequence of load and store operations: this approach does not scale well when the length of the sequence is large or symbolic, i.e., input-dependent. Hence, a symbolic executor using MemSight would still need to concretize the size of a `memset` or `memcpy` operation to avoid scalability issues, as done by angr.

3 APPROACH

There are two main challenges when considering memory-intensive operations in SE: (a) the memory addresses involved in these operations could be symbolic, and (b) the number of bytes, i.e., the size, that these operations need to manipulate could be symbolic. MemSight is designed to cope with the first challenge, however, it struggles with the second one as it still needs to move one by one each byte. MInt is designed in order to handle two memory-intensive operations (`memcpy` and `memset`) with a symbolic size, possibly scaling even in presence of large (symbolic) values.

Why `memcpy` and `memset` are interesting. In this paper, we focus our discussion on two well-known memory-intensive operations: `memcpy` and `memset`. These two primitives are often used directly, or *indirectly* through a library, by many real-world programs: for instance, the C heap allocator typically performs a `memcpy` operation when reallocating a block inside the `realloc` function and a `memset` operation when returning a zero-initialized memory block from the `calloc` function. Due to the lack of space, we do not cover in our discussion how to extend MInt to support other memory-intensive primitives offered by the C library.

Improved algorithm. Figure 2 depicts with a gray background our improvements to the original algorithm from MemSight. The global data has been extended in the following ways: (a) the concrete memory is now a mapping between a concrete address and a set of tuples (while before it was a mapping to a single tuple), (b) the memory M_{ms} tracks tuples generated by `memset` operations, and (c) the memory M_{mc} tracks tuples generated by `memcpy` operations.

```
Global data:
$M_c$ := concr. mem.  ▷ hash-table $e \rightarrow$ set of tuples $(id, a, b, e, v, t, \delta)$
$M_s$ := symbolic memory  ▷ set of tuples $(id, a, b, e, v, t, \delta)$
$M_{ms}$ := memset mem.  ▷ set of tuples $(id, e, size, v, t, \delta)$
$M_{mc}$ := memcpy mem.  ▷ set of tuples $(id, e_{src}, e_{dst}, size, t, \delta)$

function STORE($e$, $V$, $size$, $\delta$):
    $a \leftarrow min(e)$
    $b \leftarrow max(e)$
    $t \leftarrow t + 1$
    for $k = 0$ to $size - 1$ do   // $V_k$ is k-th byte of $V$
        if $a = b$ then UpdMap($M_c[a]$, $(id_c, a + k, a + k, V_k, t, \delta)$)
        else $M_s \leftarrow M_s \cup \{(id_s, a + k, b + k, e + k, V_k, t, \delta)\}$

function MEMSET($e$, $v$, $size$):
    $t \leftarrow t + 1$
    $M_{ms} \leftarrow M_{ms} \cup \{(id_{ms}, e, size, v, t, true)\}$

function MEMCPY($e_{dst}$, $e_{src}$, $size$):
    $t \leftarrow t + 1$
    $M_{mc} \leftarrow M_{mc} \cup \{(id_{mc}, e_{src}, e_{dst}, size, t, true)\}$

function LOAD($e$, $size$):
    $V = \langle\rangle$
    for $k = 0$ to $size - 1$ do
        $V_k$ = _LOAD($e + k, t$)
        $V = V \circ V_k$   // concatenation of values
    return $V$

function _LOAD($e$, $\bar{t}$):   // load of a single byte
    $a \leftarrow min(e)$
    $b \leftarrow max(e)$
    $P \leftarrow$ SEARCH($a, b, \bar{t}$)   // tuples sorted by timestamp
    $v \leftarrow 0$   // default in case of uninit. memory
    for $x \in P$ do
        if $x.id = id_c \vee x.id = id_s$ then
            $v \leftarrow ite(x.e = e \wedge x.\delta, x.v, v)$
        else if $x.id = id_{ms}$ then
            $v \leftarrow ite(x.e \leq e < x.e + x.size \wedge x.\delta, x.v, v)$
        else if $x.id = id_{mc}$ then
            $v \leftarrow ite(x.e_{dst} \leq e < x.e_{dst} + x.size \wedge x.\delta,$
                            _LOAD($e - x.e_{dst} + x.e_{src}, x.t - 1$), $v)$
    return $v$

function SEARCH($a, b, \bar{t}$):
    $P \leftarrow \emptyset$
    for $k \leftarrow a$ to $b$ do  $P \leftarrow P \cup \{x \mid x \in M_c[k] \wedge x.t \leq \bar{t}\}$
    $P \leftarrow P \cup \{x \mid x \in M_s \wedge [x.a, x.b] \cap [a, b] \neq \emptyset \wedge x.t \leq \bar{t}\}$
    $P \leftarrow P \cup \{x \mid x \in M_{mc} \wedge x.t \leq \bar{t}\}$
    $P \leftarrow P \cup \{x \mid x \in M_{ms} \wedge x.t \leq \bar{t}\}$
    return SORT_BY_INCREASING_TIMESTAMP($P$)
```

Figure 2: Pseudocode of MInt**. Lines highlighted in gray are the additional code w.r.t.** MemSight**.**

The STORE operation is the same as in MemSight with only one but significant difference on UpdMap: this procedure is now adding the tuple to the set of tuples mapped to $M_c[a]$. Hence, $M_c[a]$ keeps track of *all* concrete store operations performed over a constant address *a*, while MemSight was keeping track of only the *last* concrete store operation over *a*.

Two new primitives are exposed to handle memory-intensive operations: MEMSET and MEMCPY. MEMSET first increments the global timestamp and then stores inside M_{ms} a new tuple to represent that there was at time t a `memset` operation using a value v on (symbolic) addresses in the range $[e, e+size)$. Similarly, MEMCPY first increments the global timestamp and then stores inside M_{mc} a new tuple to represent that there was at time t a `memcpy` operation that moved bytes from (symbolic) addresses in $[e_{src}, e_{src} + size)$ to (symbolic) addresses in $[e_{dst}, e_{dst} + size)$.

The primitive LOAD is not changed, however, MInt revises its two subroutines SEARCH and _LOAD. Line 29 in SEARCH is now in charge of retrieving *all* the tuples associated with $M_c[k]$, while lines 31-32 retrieve tuples from the two new memories M_{mc} and M_{ms}. The _LOAD subroutine is where MInt differentiates significantly from MemSight. After retrieving the tuples from the different memories at line 18, _LOAD iteratively builds a (possibly nested) ite expression for the memory address e. Given a tuple x from M_{ms}, lines 23-24 build an ite expression that states that the value read from the memory is $x.v$ if $x.e \leq e < x.e + x.size$, i.e., the address e is within a memory region $[x.e, x.e + x.size)$ previously written by a memset operation using the value $x.v$.

Given a tuple x from M_{mc}, lines 25-26 instead build an ite expression that states that if the memory address e is in the interval $[x.e_{dst}, x.e_{dst} + x.size)$, i.e., if it falls in a memory region previously written by memcpy, then the return value is obtained by loading, with a recursive call to _load, the memory value which *was* in memory at the memory addresses $[x.e_{src}, x.e_{src} + x.size)$ at the time when the memcpy operation was performed (i.e., $x.t - 1$). The memory address e' that is used in the recursive call is equal to $(e - x.e_{dst})$, which is the displacement of e from $x.e_{dst}$, plus $x.e_{src}$. Hence, MInt is reasoning on the effects of a memcpy operation during a load operation by going back in time to a previous memory state identified by the timestamp $x.t - 1$. In other words, the memory state is *lazily* recovered by analyzing the history of operations over the memory. To support this *wayback machine*, MInt, differently from MemSight, has to keep track of the entire history of concrete stores in M_c.

Optimizations. Keeping track of the entire history of concrete stores may incur a large time and space overhead. MInt devises an optimization that discard old tuples in $M_c[k]$ whenever a concrete address $k \notin [x.e_{src}, x.e_{src} + x.size)$ for *all* memcpy operations tracked by M_{mc}. Additionally, given the time t' of a memcpy operation, MInt has to keep track only of the *last* store operation before t'.

4 PRELIMINARY EVALUATION

We implemented MInt in angr version 9.0. This is a binary symbolic execution framework that can analyze ELF and PE executables for the most common architectures. We remark that MInt does not make any assumption on the symbolic framework and thus could be implemented in other frameworks (even source-based ones). We now report the results of a preliminary evaluation.

Motivating examples. To validate our prototype, we have analyzed the programs presented in Figure 1 using MInt: angr is now able to reach all feasible return statements, without ever exploring unreachable code. As discussed in Section 1 and Section 2, this is not the case with the unmodified angr, even when improved with MemSight, which is unable to reach all feasible return statements and could even generate states related to unreachable code.

DARPA CGC binaries. To further evaluate MInt, we have considered 100 binaries taken from the qualification rounds of the DARPA Cyber Grand Challenge [26]. All these binaries are making use of memcpy and memset, supporting our claim that these two operations are very common in several programs. However, since not all memset and memcpy operations may lead to scalability issues, we restrict our discussion to the 9 binaries (CROMU_{18, 32, 33, 38} and NRFIN_{16, 22, 24, 26, 34}) out of 100 applications where we were able to identify memset and memcpy operations with symbolic arguments during a 2-hour symbolic exploration with angr. Given these 9 applications, we have repeated the symbolic exploration considering different memory models for angr: (a) angr with its standard memory model, which may concretize symbolic pointers spanning large memory regions to favor scalability, (b) angr*, a variant of angr that considers *all* possible values admissible for a symbolic pointer, (c) MInt, i.e., our prototype, and (d) MemSight, i.e., angr using the original memory model from MemSight. To make the comparison meaningful, the exploration is performed using a breadth-first search (BFS), proceeding in rounds where a new round at depth d starts only when all states in the previous round $d - 1$ have been explored (evaluating one basic block for each state). Given an exploration at depth d, we have tracked: the number n_{ops} of memory-intensive operations with symbolic arguments, the number of states, the time (seconds) took by the exploration, and the number of edges in the interprocedural control-flow graph (ICFG) traversed during the exploration.

BENCHMARK		ANGR	ANGR*	MEMSIGHT	MInt
CROMU_18	# STATES	767	failed	failed	1531
$n_{ops} = 686$	TIME (SECS)	3048	failed	failed	7049
$d = 3023$	# COV. EDGES	272	failed	failed	280
CROMU_32	# STATES	10	failed	failed	14
$n_{ops} = 32$	TIME (SECS)	155	failed	failed	6719
$d = 3477$	# COV. EDGES	147	failed	failed	152
NRFIN_16	# STATES	34	failed	failed	35
$n_{ops} = 1$	TIME (SECS)	2118	failed	failed	2150
$d = 4101$	# COV. EDGES	394	failed	failed	397
NRFIN_24	# STATES	1670	failed	failed	2026
$n_{ops} = 1063$	TIME (SECS)	7166	failed	failed	2354
$d = 3157$	# COV. EDGES	205	failed	failed	245

Table 1: Experiments on DARPA CGC binaries. n_{ops} is the number of memory-intensive operations with symbolic args, while d is the depth of the symbolic exploration.

On application NRFIN_22, all memory models behave quite similarly. On 4 applications (CROMU_{33, 38} and NRFIN_{16, 26}), MInt generates more states than angr but shows overall similar results to MemSight. On the remaining 4 binaries, MInt seems to show some valuable advantages with respect to the competitors. Table 1 reports the tracked statistics for these binaries: we consider the maximum depth d reached by both angr and MInt. angr* and MemSight, which similarly to MInt aim at an exhaustive symbolic exploration, fail to reach the considered round. This is interesting as it supports our claim that existing memory models that target accurate reasoning on symbolic memory accesses cannot scale in presence of memory-intensive operations over symbolic data.

On CROMU_18, MInt generates 764 more states than angr, covering 6 additional ICFG edges. Although the improvement in coverage may seem marginal, it is actually interesting since CGC binaries have on average less than 600 edges in their ICFG. The analysis time scales nicely: double the time taken by angr, which however only explores half of the states considered by MInt.

On CROMU_32, MInt generates more states and covers more edges than angr, however, the analysis time seems to scale poorly: MInt does not force angr to concretize the size of some memory blocks, allowing the symbolic engine to reason symbolically on the heap allocator implementation, generating some hard-to-solve queries and several extremely *large* symbolic accesses.

On NRFIN_16, MInt slightly improves the results from angr, generating one additional state and covering 3 additional edges. This is not surprising as only a single memory-intensive operation has been observed during the symbolic exploration.

Finally, on NRFIN_24, MInt shines against the competitors, generating more states (+20%) and covering more edges (+19%) than angr, while also drastically reducing the analysis time (-67%). The improvement in the analysis time is due to the large number of memory-intensive operations ($n_{ops} = 3157$).

5 CONCLUSIONS

In this paper, we have presented MInt, a memory model for symbolic execution that is aware of two memory-intensive operations, memcpy and memset, respectively. Our experiments have shown that from 100 binaries taken from the DARPA CGC, all of them make use of these two operations. On 9 binaries out of 100, these operations may lead to scalability issues, as shown by our motivating examples. When analyzing these binaries with MInt, the symbolic exploration is able to reach code points that are missed by the standard angr and by MemSight.

As future work, we believe that MInt could be extended towards two directions: (a) add support for additional memory-intensive primitives (e.g., memchr), and (b) make the memory model aware even of other types of operations, such as heap allocations.

ACKNOWLEDGMENTS

This work is supported by the ERC Advanced Grant WhiteMech (No. 834228) and by the European Union's Horizon 2020 research and innovation programme under grant No. 830892, project SPARTA.

REFERENCES

[1] Marco Angelini, Graziano Blasilli, Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, Simone Lenti, Simone Nicchi, and Giuseppe Santucci. 2019. SymNav: Visually Assisting Symbolic Execution. In *Proc. of the 16th IEEE Symposium on Visualization for Cyber Security (VizSec '19)*. https://doi.org/10.1109/VizSec48167.2019.9161524

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proc. of the 36th Int. Conf. on Soft. Eng. (ICSE '14)*. 1083–1094. https://doi.org/10.1145/2568225.2568293

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Assisting Malware Analysis with Symbolic Execution: A Case Study. In *Proc. of the 1st Int. Conf. on Cyber Security Cryptography and Machine Learning (CSCML '17)*. https://doi.org/10.1007/978-3-319-60080-2_12

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computer Surveys* 51, 3, Article 50 (5 2018), 39 pages. https://doi.org/10.1145/3182657

[5] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Memory models in symbolic execution: key ideas and new thoughts. *Soft. Testing, Verification and Reliability* (2019). https://doi.org/10.1002/stvr.1722

[6] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution. In *Cyber Security Cryptography and Machine Learning (CSCML '19)*. https://doi.org/10.1007/978-3-030-20951-3_12

[7] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://doi.org/10.1109/ICSE43902.2021.00071

[8] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. FUZZOLIC: mixing fuzzing and concolic execution. *Computers & Security* (2021). https://doi.org/10.1016/j.cose.2021.102368

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *8th USENIX Conf. on Operating Systems Design and Implem. (OSDI '08)*. 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS '06)*. https://doi.org/10.1145/1180405.1180445

[11] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. https://doi.org/10.1109/SP.2012.31

[13] Emilio Coppa, Daniele C. D'Elia, and Camil Demetrescu. 2017. Rethinking pointer reasoning in symbolic execution. In *Proc. of the 32nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '17)*. 613–618. https://doi.org/10.1109/ASE.2017.8115671

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *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)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[15] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54 (2011), 69–77. https://doi.org/10.1145/1995376.1995394

[16] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise Pointer Reasoning for Dynamic Test Generation. In *Proc. 18th Int. Symp. on Soft. Test. and Analysis (ISSTA '09)*. https://doi.org/10.1145/1572272.1572288

[17] Stephan Falke, Florian Merz, and Carsten Sinz. 2014. Extending the Theory of Arrays: memset, memcpy, and Beyond. In *Verified Software: Theories, Tools, Experiments*, Ernie Cohen and Andrey Rybalchenko (Eds.). 108–128.

[18] Stephan Falke, Carsten Sinz, and Florian Merz. 2013. A Theory of Arrays with set and copy Operations. In *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories (EPiC Series in Computing, Vol. 20)*, Pascal Fontaine and Amit Goel (Eds.). 98–108. https://doi.org/10.29007/q58t

[19] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays *(Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07))*. 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[20] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. 774–784. https://doi.org/10.1145/3338906.3338936

[21] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. 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))*. 337–348. https://doi.org/10.1145/2150976.2151012

[22] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[23] Corina S. Păsăreanu and Willem Visser. 2009. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal on Software Tools for Technology Transfer* 11 (2009), 339–353. https://doi.org/10.1007/s10009-009-0118-1

[24] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. https://doi.org/10.1109/SP.2010.26

[25] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*. 138–157. https://doi.org/10.1109/SP.2016.17

[26] Jia Song and Jim Alves-Foss. 2015. The DARPA Cyber Grand Challenge: A Competitor's Perspective. *IEEE Security Privacy* 13, 6 (2015), 72–76. https://doi.org/10.1109/MSP.2015.132

[27] David Trabish and Noam Rinetzky. 2020. Relocatable Addressing Model for Symbolic Execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. 51–62. https://doi.org/10.1145/3395363.3397363

[28] Marek Trtík and Jan Strejček. 2014. Symbolic Memory with Pointers. In *Proceedings of 12th International Symposium on Automated Technology for Verification and Analysis (ATVA '14)*. 380–395. https://doi.org/10.1007/978-3-319-11936-6_27