Inuring: Live Attacker-Guided Repair

Eric Schulte
Suan Yong
Dave Melski
eschulte@grammatech.com
suan@grammatech.com
melski@grammatech.com
GrammaTech, Inc
Ithaca, New York

ABSTRACT

We present *inuring*, an attack-guided repair method for software vulnerabilities in n-variant systems. N-variant systems detect attacks that cause divergence in variant behavior, converting severe vulnerabilities (such as those that enable remote code execution) into less severe denial-of-service vulnerabilities. Inuring is a general technique for n-variant systems that uses information gleaned from an attack to perform a "live" field repair of the underlying vulnerability, thereby obviating the denial-of-service attack. We present a case study of the use of inuring to protect against a powerful class of memory-corruption exploits in the Apache web server. Our demonstration leverages *dappling*, a new technique for provably secure memory layout in n-variant systems. With inuring and dappling we are able to guarantee strong protection and remediation for a class of *write-what-where* vulnerabilities in n-variant systems. Our case study illustrates the efficacy and efficiency of these techniques.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Inuring, n-variant, memory safety

ACM Reference Format:
Eric Schulte, Suan Yong, and Dave Melski. 2019. Inuring: Live Attacker-Guided Repair. In *3rd Workshop on Forming an Ecosystem Around Software Transformation(FEAST'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3338502.3359761

1 INTRODUCTION

Modern society increasingly runs on software, and our vulnerability to software attacks is increasing commensurately. Unfortunately software security is asymmetric: not only do attackers get to go second, but defenders have to be right all of the time while attackers only have to be right once. We introduce an *inuring* method in which deployed software automatically hardens itself in response to attack. Inuring not only stops attacks, but also permanently protects the software against subsequent attempts to exploit the same weakness. We argue that inuring is a general technique that is applicable to many attack surfaces in n-variant systems.

Memory errors remain a rich source of exploits for attackers. A decline in other attack options means that they are even gaining in popularity (cf. increased interest in *data only attacks* [6]): for example, classic control flow attacks such as ROP are largely mitigated in modern architectures due to advances in control flow protection.

We present a case study using inuring to defend against memory exploits in an n-variant system. This illustration introduces a novel n-variant memory layout technique called *dappling* which provides complete spatial memory protection for dappled memory. This work makes the following contributions.

C1. *Dappling*, a secure and efficient method of laying out memory across multiple program variants to prevent absolute and offset attacks (§ 2).
C2. *Inuring*, a generic method for attack-guided repair of n-variant systems (§ 3).
C3. An application of inuring to detect and repair memory violations in n-variant systems (§ 3.1).
C4. An inuring case study and evaluation (§ 4).

1.1 Background and Related Work

1.1.1 N-Variant Systems. An "n-variant system" [3] is a system in which multiple versions of a program are run in unison. All input is multiplexed to the variants, and the responses from all variants are unified and checked for unanimous agreement before the system responds. The harness for an n-variant system may be implemented in the kernel or in user space. The variant input and output may be multiplexed and unified around system calls. N-variant systems can be a convenient way to leverage the additional cores provided by modern systems to provide additional security for safety-critical software. Each variant should differ in the details relevant to attack surface, such as memory layout. For an attack to land, it must simultaneously corrupt all variants in an analogous manner so that they continue to all give the same responses. For example, an attack would have to corrupt the same function pointer in each variant to point to corresponding addresses in each variant. Given diversity among the variants, attempted exploits are likely to result in divergence.

Examples of n-variant systems include the following.

FEAST'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6834-6/19/11…$15.00
https://doi.org/10.1145/3338502.3359761

```
            low memory        high memory
variant 1  XXXXXXXXXXXXXXXXXX..................
variant 2  ..................XXXXXXXXXXXXXXXXXX
```

Figure 1: "Checkerboard" layout with no absolute memory overlap between variants.

MvArmor (and MemSentry). MvArmor[1] is a multi-variant execution system using hardware-assisted process virtualization [7]. MemSentry[2] is an MvArmor follow-on which uses hardware to isolate sensitive regions.

Varan. Varan [5] focuses on performant multi-version execution without requiring kernel modifications.

FreeDA. FreeDA [11] runs multiple incompatible dynamic analysis tools via multi-variant execution.

Bunshin. Bunshin [16] distributes checks across variants to maximize compatibility and efficiency.

Buddy. Buddy [9] provides probabilistic protection against memory disclosure.

1.1.2 Memory Errors. We identify three classes of spatial memory errors.

Buffer overflow. The most common memory error is a buffer overflow. In a buffer overflow, the code responsible for accessing a region of memory ('buffer') doesn't perform proper bounds checking and may read or write past the end of the region. When exploited this may give an attacker read or write access to whatever program data happens to immediately follow the buffer in program memory.

Offset attack. An offset attack is a generalization of a buffer overflow that targets an indexed memory access such as the following.

```
... *(base + index * scale) ...
```

In a buffer overflow, the attacker leverages the fact that the program may increment (or decrement) the *index* to values that are outside of the buffer pointed to by *base*. In an offset attack, the attacker has the ability to set *index* and possibly *scale* to arbitrary values of their chosing. This provides the attacker with the ability to access memory arbitrary distances from the *base* of the buffer, without having to traverse all intervening memory. This bypasses defenses against buffer overflows that place special "guard" values immediately following every buffer in memory and raising an error when a guard value is accessed [10, 12].

Write-What-Where. A write-what-where attack targets an instruction that dereferences an improperly handled address to update a memory location.

```
*(base + index * scale) = value
```

The attacker takes control of both the address and the *value* that is written. A write-what-where attack is a flexible, robust primitive for building attacks.

A write-what-where attack may or may not be an offset attack, depending on what components of the address are under attacker control. In an n-variant system offset attacks are especially threatening, because many diversification techniques do not change the relative offsets between program elements. Each variant may use a different *base* value, but the attack is indifferent, because the same *index* values will access the same code or data in each variant.

[1] https://github.com/vusec/mvarmor
[2] https://github.com/vusec/memsentry

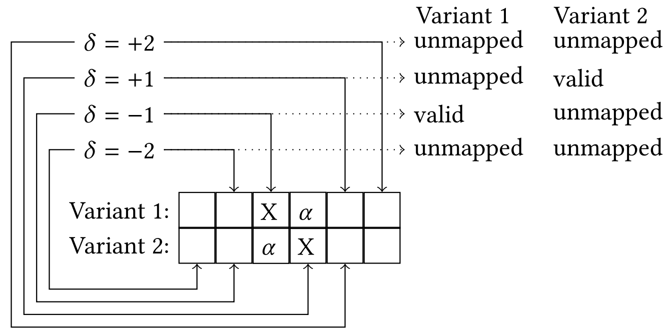

Figure 2: Offset attacks against a dappled layout. $\forall \alpha \nexists$ any offset δ s.t. $\forall$ variant $\alpha + \delta$ is a valid address. A two-variant and two-object layout is shown.

Consider the simple "checkerboard" memory layout in Figure 1, where "`X`"s represent program code and data and "`.`"s represent unmapped memory. Any write-what-where attack against an absolute address value will hit unmapped memory in one variant as there are no addresses mapped in every variant. However, assuming the program code and data are laid out identically across both variants a simple buffer overflow or offset attack would still work because each variant will perform the memory calculation with its own suitable value for *base*.

2 DAPPLING

Diverse memory layouts across n-variant systems may be used to protect against spatial memory vulnerabilities. However, as described in the previous section, existing diversification techniques may be insufficient to protect against offset attacks.

Dappling provides provably complete protection against spatial memory attacks in the dappled memory. An SMT solver is used to synthesize layouts of program data across all variants in a manner which is formally guaranteed to preclude any spatial memory errors (including buffer-overflow, write-what-where, and offset attacks) *and* to be maximally space-efficient. The use of existing kernel-level page faults avoids the need for explicit application-level checks around memory accesses. Dappling at sub-page granularity is possible if application-level checks are used as in AddressSanitizer or Light-Weight Bounds Checking (LWBC) [10, 12]. Temporal memory errors such as use-after-free are not addressed by dappling.

Specifically, dappled layouts ensure that $\forall$ program object α $\nexists$ any nonzero offset δ s.t. $\forall$ variant in the n-variant system the address $\alpha + \delta$ is valid. Although such layouts are easily written by hand and checked for small numbers of variants and objects (e.g., the 2×2 layout in Figure 2) they quickly become difficult to identify and check. The key technical insights of this work are the use of SMT solvers to efficiently synthesize maximally efficient layouts for given numbers of variants and objects (§ 2.1), and the recursive

```
Variant 1:  01234567.8.9.ABCDEF
Variant 2:  5FDA.B38E.26.79104C
Variant 3:  47690.1DFCA2.3.E8B5
Variant 4:  CE5804.FA3B.D.62971
```

(a) Dense layout of 16 objects over 4 variants in 19 memory locations.
Every object and memory location has the same size.

```
Variant 1:   .........01234567.8.9.ABCDEF
Variant 2:   .......5FDA.B38E.26.79104C...
Variant 3:   47690.1DFCA2.3.E8B5.........
Variant 4:   ....CE5804.FA3B.D.62971......
```

(b) Layout from 3a aligned on '3', visualizing the offsets from object 3. Note every offset, column, is unmapped or guarded (shown as "`.`") in one variant. Furthermore this property holds when the layouts are re-aligned on any object from `0`-`F`.

Figure 3: An example dense dappled layout.

re-application of layouts at increasing orders of magnitude of scale to handle multiplicatively more objects (§ 2.3).

2.1 SMT-Encoding

For any given number of variants, size of memory, and number of objects of equal size, it is possible to efficiently encode the constraints of Figure 2 using the theory of fixed width bit vectors.

Appendix A shows such an encoding expressed using the Common Lisp CL-SMT-LIB package[3] for encoding constraints in SMT-LIB2[4] [1]. This encoding works by representing layouts as bit-vectors. Then for every object in the layout which is the base of a potential offset attack it is asserted that when the bit-vectors are aligned on that object and zero-padded their bitwise-and is 0. This ensures that every position is 0, i.e. unmapped or guarded, in at least one variant.

The SMT solver will either return `unsat` (meaning there is no arrangement with this property) or it will return the satisfying model: a dappled layout of objects across variants. An example SMT-generated layout is shown in Figure 3a. Figure 3b shows this same layout aligned on the object labeled "3." There is at least one unmapped or guarded location in "column:" that is, at every possible nonzero offset from the aligned object. This property holds for every alignment on every object in the layout.

2.2 Dense Dappled Layouts

To find all maximally dense layouts, we begin with a set number of variants, a set range (number of memory slots), and two objects, and run the SMT solver with increasing number of objects until `unsat` is returned for some number of objects N. This yields the densest packing of $N-1$ objects in that range. Repeatedly running until `unsat` and ratcheting up the range yields the densest layout for each range across the given number of variants. The sizes of these densest possible layouts for two through eight variants are shown

[3] https://github.com/grammatech/cl-smt-lib
[4] http://smtlib.cs.uiowa.edu/

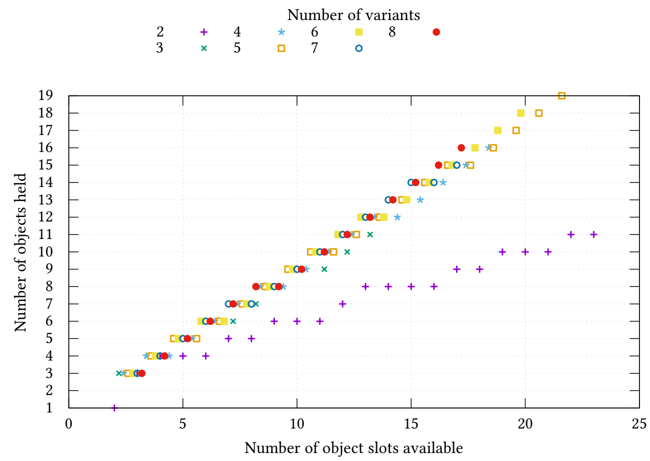

Figure 4: The sizes of the densest possible dappled layouts by number of variants and number of objects.

in Figure 4. The SMT queries required to identify these layouts quickly become expensive (taking many days of time to compute the largest layouts shown). We have been unable to calculate the growth of minimal layout size as a function of the number of objects and variants and we are unsure whether or not a closed-form relationship exists.

2.3 Recursive Dappling

As shown in Figure 4, the largest layouts house orders of magnitude fewer objects than are used by a typical program. Complexity of layout calculation is exponential in the number of objects, so even years of solver time would add only a couple of points to this graph.

We can scale these layouts to real programs by recursively applying the layouts as shown in Figure 5. To perform this "recursive dappling" we repeatedly apply a layout as many times as necessary until all program objects are housed: a total of $\frac{|objects|}{N}$ applications, where $|objects|$ is the number of program objects and N is the number of objects held by the layout. We then treat each application as an atomic object and recursively dapple them. Dappling assumes layouts are surrounded by unmapped memory of size equal to the range, and recursive dappling may pack applications adjacently, so we append unmapped memory of range size to one side of each application before recursive dappling. We then recursively dapple the resulting applications into $\frac{|objects|}{N^2}$ total higher-level applications. This process continues until only a single application is required to dapple all sub-applications. In this way even very large programs may be dappled using modestly sized layouts.

2.4 Limitations and Extensions

Limitations. The dappled layouts shown thus far can be directly applied to all statically allocated program code and data. Dappled layouts do not protect against intra-structure memory errors, only errors between independent objects. Dappled layouts cannot be used directly to dapple dynamic heap and stack data.

Extensions. For both stack and heap one could first select a maximum amount of space which may be consumed dynamically and

```
scale |                        layout                        | objs | range
-----------------------------------------------------------------------------
  4   | 12..34......56..78..................9A..BC......DE..FG |  16  |  54
      | GF..ED......CB..A9..................87..65......43..21 |      |
      |                                                        |      |
  3   | 12..34......56..78                                     |  8   |  18
      | 87..65......43..21                                     |      |
      |                                                        |      |
  2   | 12..34                                                 |  4   |  6
      | 43..21                                                 |      |
      |                                                        |      |
  1   | 12                                                     |  2   |  2
      | 21                                                     |      |
```

Figure 5: Recursive application of a 2-variant 2-object dappling layout at increasing scales.

then generate a dappled layout to accommodate this much space. Then when memory is allocated and freed (either by the heap allocator or by stack growth and shrinkage) the memory would be consumed from and returned to each variant's dappled layout in the order of the objects in that layout. This would be a refinement of the "dense/sparse" heap memory layout strategy presented in MvArmor [7] which could be viewed as a degenerate space-inefficient form of heap dappling.

Such a method of dynamic dappling is not implemented in our prototype and is not included in our case study. Implementation would require modification to the heap allocator and to the mechanism by which activation records are added to and removed from the stack. It could cause excessive runtime overhead due to the extra runtime cost of referencing and maintaining the layout on every allocation. It could also cause excessive memory pressure if most heap allocated objects require independent memory pages.

3 INURING

Inuring is a technique to automatically immunize an n-variant system in response to an attempted attack. Inuring requires three components:

(1) A structured diversification that guarantees that an attempted exploit will (a) cause a divergence and (b) identify the location of the vulnerability.
(2) An ability to install consensus voting in the variants such that future attempted exploits can be detected prior to divergence.
(3) A suitable replacement action to take in place of executing the original, vulnerable logic.

N-variant systems convert severe vulnerabilities, such a those that enable remote-code execution, into less severe denial-of-service vulnerabilities. Inuring overcomes this limitation: after a vulnerability is inured, attackers cannot use the inured vulnerability to force divergence.

We demonstrate inuring for an important class of memory-safety vulnerabilities that is frequently expensive and difficult to defend. We believe the technique is applicable to other vulnerability classes where the three obligations listed above can be met.

3.1 Inuring Against Memory Exploits

We guarantee divergence on attempted offset attacks by dappling objects and employing a memory-checking technique similar to LWBC or AddressSanitizer [10, 12] (which do not protect against offset attacks on their own). An attack leveraging an offset from one object to another will be detected in at least one variant, either by causing a dereference of unmapped memory or by attempting a memory access that is forbidden by the memory-checking technique. In either case the instruction of the bad memory access is identified satisfying inuring requirement (1).

Every potentially unsafe memory access is checked in every variant. However, dappling only guarantees that a check will catch an invalid access in at least one variant. The other variants may clear the access as safe, because it does not access a guarded memory zone in those variants. When an attack is detected (in some variant), we force divergence and a system reset before any corrupted variants can commit a persistent action, such as writing to disk.

To automatically repair the vulnerability, we modify the check on the vulnerable instruction in each variant to require consensus from all variants that the access is safe before proceeding. We call this technique *consensus voting*. Consensus voting requires support from the n-variant system. Prior to inuring, the code around an unsafe memory access might look like the following:

```
if (accesses_guard_zone(p))
  diverge();
... *p ...     // Dereference of potentially unsafe address p.
```

After inuring, the code behaves as follows:

```
if (nsys_consensus(accesses_guard_zone(p)))
    ... *p ... // Original instruction.
else
    ...        // Replacement action.
```

In this code, `nsys_consensus` is a call to the n-variant system that returns true (indicating the access is safe in all variants) if and only if the corresponding calls pass in all variants.

Consensus voting converts the guarantee that an attempted attack will be detected in some variant into a guarantee that all variants will avoid the attack. It is too expensive to implement proactively because it requires synchronization between variants. Once a vulnerability has been verified by an attempted attack, the additional cost in justified.

Inuring requires an alternative to exercising the original, vulnerable code. Many replacement actions are available, including:

A1. Skip write operations and replace read operations with the constant value zero. This approach of using "null actions" was introduced by Recovery Shepherding [8].
A2. Configure an application-specific replacement action. For example, server applications are often capable of recovering from dropped or faulty connections. When an attack is detected in handling a connection, the replacement action could drop the connection. This approach is a variant of error virtualization [13].
A3. Alert an existing protection and response system. There are many commercial systems that perform Software Information and Event Management (SIEM). These systems may be configured with knowledge of the mission or goals of the system and specific responses to attack.
A4. Intentionally delay the performance of the system [14, 15]. This strategy was first used in intrusion detection systems.

Table 1: Pseudo-code versions of unsafe memory accesses inured with the "null" replacement strategy A1 described in § 3.

	Original	Mem-safety check	Inured check

Unsafe Write	`*p = x;`	`tmp = p;`
		`if (guard_bad(tmp))`
		`  abort();`
		`*tmp = x;`
			`tmp = p;`
			`imbad = guard_bad(tmp);`
			`anybad = consensus(imbad);`
			`if (!anybad)`
			`  *tmp = x;`

Unsafe Read	`y = *p;`	`tmp = p;`
		`if (guard_bad(tmp))`
		`  abort();`
		`y = *tmp;`
			`tmp = p;`
			`imbad = guard_bad(tmp);`
			`anybad = consensus(imbad);`
			`y = anybad ? 0 : *tmp;`

4 CASE STUDY

We implemented an inuring prototype using GrammaTech's CodeSurfer® binary rewriting system to generate N variants of an example binary for execution under a Multi-Variant Execution Environment (MVEE) named RAVEN [2]. Specifically, we inure the Apache web server and demonstrate that the hardened server (i) detects and reports memory-safety violations the first time they occur, (ii) patches exploited locations with alternate inured code, and (iii) on subsequent violations at the same site, continues execution without violating memory safety and without divergence or restart.

Starting with a version of Apache 2.4.17 which had been seeded with multiple vulnerabilities we did the following.

1. (1) Create two variants, each instrumented to perform memory-safety checks akin to Address Sanitizer [10, 12]. Memory-safety checks are applied to every dereference that cannot be statically determine to be safe.
2. (2) Pre-generate "inured" versions of every basic block (i.e. sequence of instructions with straight-line control flow) that contains an unsafe memory dereference. These inured basic blocks are automatically adapted from the originals by applying the "null action" strategy A1 described in § 3. Inured blocks are added to the binary but are not targets of control flow.
3. (3) Dapple the global objects across the variants. Dappling works for objects of uniform size, so this requires first grouping the objects by size, then padding all objects in each group up to the size of the largest object in the group. Dapple each group using a memory slot sizes equal to the largest object in the group. Then recursively dapple the resulting applications.
4. (4) Start the two variants under the MVEE and run non-malicious but rigorous JMeter [4] tests to confirm runtime efficiency and that no divergence is reported.
5. (5) Run a proof-of-vulnerability (POV) input designed to exploit one of the seeded vulnerabilities. This will trigger a memory fault in at least one variant, causing the MVEE to diverge. On divergence inuring takes place automatically:
   - (a) Identify the basic block B in which the memory safety violation occurred.
   - (b) Path both variants to replace the beginning of block B with a jump instruction, i.e. a trampoline, to the corresponding inured basic block B_{inured}.
   - (c) Restart Apache, now running inured forms of both of the original variants.
6. (6) Repeatedly run the POV input against the inured server. These subsequent attacks cause the inured blocks to run consensus vote as shown in Table 1. The consensus vote fails on malicious input causing the replacement action to be run. Instead of divergence or restart, the only impact on behavior is the minimal impact of the replacement action and the modest slow down of the consensus vote.

5 CONCLUSION

We present *dappling* and *inuring*. Dappling is a maximally space-efficient technique of memory layout in n-variant systems that is provably secure against spatial memory errors. Inuring is a method of attack-guided repair in n-variant systems. Inuring leverages attacker input to identify vulnerabilities through divergence of an n-variant system. This divergence then triggers the application of automated general repair techniques leveraging the use of consensus voting between variants. While slower than normal execution, consensus voting avoids the extreme impacts of divergence which could otherwise result in a denial-of-service. Instead of diverging, the inured system may respond to attack by skipping vulnerable behavior, error virtualization, reporting to a SIEM system, or taking application-specific actions. Inuring is a general technique to mitigate the denial-of-service attacks that force repeated divergence of an n-variant systems.

6 ACKNOWLEDGMENTS

This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-15-C-0113 and the Office of Naval Research under contract No. N68335-17-C-0700. The views opinions findings and conclusions or recommendations contained herein are those of the authors and should not be interpreted as necessarily representing the official views policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA); or its Contracting Agent, the U.S. Department of the Interior, Interior Business Center, Acquisition Services Directorate, Division III, or the Office of Naval Research. We thank Kudu Dynamics for seeded Apache vulnerabilities, and Amy Gale for her comments on an earlier version of this paper.

REFERENCES

[1] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Déharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krsti?, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, and Jochen Hoenicke. The smt-lib standard: Version 2.0 draft. Technical report, 2010.

[2] Michele Co, Jack W Davidson, Jason D Hiser, John C Knight, Anh Nguyen-Tuong, Westley Weimer, Jonathan Burket, Gregory L Frazier, Tiffany M Frazier, Bruno Dutertre, et al. Double helix and raven: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, page 17. ACM, 2016.

[3] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[4] Emily H Halili. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[5] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *ACM SIGPLAN Notices*, volume 50, pages 339–353. ACM, 2015.

[6] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[7] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[8] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via Recovery Shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 227–238, New York, NY, USA, 2014. ACM.

[9] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[10] Hasabnis Niranjan, Misra Ashish, and R.Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO), pages 135–144, San Jose, California, 2012. ACM. 2259034.

[11] Luís Pina, Anastasios Andronidis, and Cristian Cadar. Freeda: deploying incompatible stock dynamic analyses in production via multi-version execution. *System*, 9(10):11, 2018.

[12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[13] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2009.

[14] Anil Somayaji and Stephanie Forrest. Automated response using system-call delay. In *Usenix Security Symposium*, pages 185–197, 2000.

[15] Matthew M Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, pages 61–68. IEEE, 2002.

[16] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: compositing security mechanisms through diversification. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 271–283, 2017.

A CL-SMT-LIB2 DAPPLING CONSTRAINTS

```
(defun dapple (range vars objs)
  "Return a query for a satisfiable layout in RANGE across VARS holding OBJS.
The order of OBJS is allowed to permute between variants. A
satisfiable layout will ensure that no offset exists which when added
to the location of an object in each variant will land in another
object in each variant.
* RANGE is the range in which to layout the objects.
* VARS is the number of variant layouts to consider.
* OBJS is the number of objects which should be held in each variant layout."
  (flet ((bit-n (n) (read-from-string (format nil "|bit-~d|" n)))
         (obj-n (n) (read-from-string (format nil "|obj-~d|" n)))
         (bvn (n) (read-from-string (format nil "|bv~d|" n)))
         (vn (n) (read-from-string (format nil "|v~d|" n)))
         (vpn (n o) (read-from-string (format nil "|vp~d-~d|" n o))))
    #!`((set-option :produce-models true)
        (set-logic QF_BV)

        (define-fun hamming-weight ((bv (_ BitVec ,RANGE))) (_ BitVec ,RANGE)
          ,(REDUCE (LAMBDA (ACC N)
                     `(bvadd ,ACC ((_ zero_extend ,(1- RANGE))
                                   ((_ extract ,N ,N) bv))))
                   (LOOP :FOR I :FROM 1 :BELOW RANGE :COLLECT I)
                   :INITIAL-VALUE
                   `((_ zero_extend ,(1- RANGE)) ((_ extract 0 0) bv))))

        (define-fun member ((index (_ BitVec ,RANGE)) (bv (_ BitVec ,RANGE)))
            Bool
          (not (= (_ bv0 ,RANGE) (bvand bv index))))

        (define-fun left-hamming-weight ; Counts INDEX from left to right.
            ((index (_ BitVec ,RANGE)) (bv (_ BitVec ,RANGE))) (_ BitVec ,RANGE)
          (hamming-weight
           (bvand bv (bvnot (bvsub (bvshl index (_ bv1 ,RANGE))
                                   (_ bv1 ,RANGE))))))

        (define-fun right-hamming-weight
            ((index (_ BitVec ,RANGE)) (bv (_ BitVec ,RANGE))) (_ BitVec ,RANGE)
          (hamming-weight (bvand bv (bvsub index (_ bv1 ,RANGE)))))

        (define-fun bit-1 ((bv (_ BitVec ,RANGE))) (_ BitVec ,RANGE)
          (bvand bv (bvneg bv)))

        ,@(LOOP :FOR N :FROM 1 :TO RANGE :COLLECT
             `(define-fun ,(BIT-N (1+ N)) ((bv (_ BitVec ,RANGE)))
                  (_ BitVec ,RANGE)
                (,(BIT-N N) (bvand bv (bvsub bv (_ bv1 ,RANGE))))))

        (define-fun index-bit ((index (_ BitVec ,RANGE)) (bv (_ BitVec ,RANGE)))
            (_ BitVec ,RANGE)
          ,(REDUCE (LAMBDA (ACC N)
                     `(ite (= index (_ ,(BVN (- RANGE N 2)) ,RANGE))
                           (,(BIT-N (- RANGE (1+ N))) bv)
                           ,ACC))
                   (LOOP :FOR I :BELOW (1- RANGE) :COLLECT I)
                   :INITIAL-VALUE `(,(BIT-N RANGE) bv)))
        ;; To allow for permutation each variant also has an
        ;; associated "permutation matrix" which may be used to map
        ;; each index to another valid index.  The matrix for N
        ;; objects will be N N-length bit vectors each of which has a
        ;; single 1, the 1 in the ith vector at the jth place means
        ;; that the permutation matrix maps i to j.
        (define-fun permute ((index (_ BitVec ,RANGE))
                             ,@(ITER (FOR OBJ BELOW OBJS)
                                     (COLLECT
                                         `(,(OBJ-N OBJ) (_ BitVec ,RANGE)))))
            (_ BitVec ,RANGE)
          (let ((my-index-bit (bvshl (_ bv1 ,RANGE) index)))
            ,(REDUCE (LAMBDA (ACC N)
                       `(ite (=  my-index-bit ,(OBJ-N N))
                             (_ ,(BVN N) ,RANGE)
                             ,ACC))
                     (REVERSE (ITER (FOR N BELOW (1- OBJS)) (COLLECT N)))
                     :INITIAL-VALUE `(_ ,(BVN (1- OBJS)) ,RANGE))))

        (define-fun left-zeros ((index (_ BitVec ,RANGE)))
            (_ BitVec ,(* 2 RANGE))
          ,(LABELS
               ((LEFT-ZEROS (SIZE OFFSET)
                  (LET ((MID (+ OFFSET (FLOOR (/ SIZE 2)))))
                    (IF (<= SIZE 1)
                        `(_ ,(BVN (- RANGE (+ OFFSET SIZE))) ,(* 2 RANGE))
                        `(ite (bvugt index
                                     (_ ,(BVN (FLOOR (EXPT 2 (1- MID))))
                                        ,RANGE))
                              ,@(LIST
                                 (LEFT-ZEROS (CEILING (/ SIZE 2))
                                             (+ OFFSET (FLOOR (/ SIZE 2))))
                                 (LEFT-ZEROS (FLOOR (/ SIZE 2)) OFFSET)))))))
             (LEFT-ZEROS RANGE 0)))

        (define-fun centered ((index (_ BitVec ,RANGE)) (bv (_ BitVec ,RANGE)))
            (_ BitVec ,(* 2 RANGE))
          (bvshl ((_ zero_extend ,RANGE) bv) (left-zeros index)))

        ,@(ITER (FOR VAR BELOW VARS)
                ;; Variant I.
                (COLLECT `(declare-const ,(VN VAR) (_ BitVec ,RANGE)))
                (COLLECT `(assert (= (_ ,(BVN OBJS) ,RANGE)
                                     (hamming-weight ,(VN VAR)))))
                ;; Permutation Matrix for Variant I.
                (UNLESS (= VAR 0)
                  (APPENDING
                   (ITER (FOR OBJ BELOW OBJS)
                         (COLLECT
                             `(declare-const ,(VPN VAR OBJ) (_ BitVec ,RANGE)))
                         (COLLECT
                             `(assert
                               (or ,@(ITER (FOR N BELOW OBJS)
                                           (COLLECT
                                               `(= (_ ,(BVN (EXPT 2 N)) ,RANGE)
                                                   ,(VPN VAR OBJ)))))))))
                  (COLLECT
                      `(assert
                        (= (_ ,(BVN (1- (EXPT 2 OBJS))) ,RANGE)
                           ,(REDUCE (LAMBDA (ACC N)
                                      `(bvor ,(VPN VAR N) ,ACC))
                                    (REVERSE
                                     (ITER (FOR N BELOW (1- OBJS)) (COLLECT N)))
                                    :INITIAL-VALUE
                                    (VPN VAR (1- OBJS))))))))
        ;; For BV of length N.
        ;;
        ;; For every object O
        ;;   For every layout L
        ;;     create the 2*N BV with L shifted left by (- N (position O))
        ;;     assert bvand across all layouts == 0 aside from the center bit
        (assert
         (and
          ,@(LABELS ((ALIGNED (OBJ VAR)
                       (LET ((CENTERED
                               (IF (ZEROP VAR)
                                   `(centered (index-bit (_ ,(BVN OBJ) ,RANGE)
                                                         ,(VN VAR))
                                              ,(VN VAR))
                                   `(centered
                                     (index-bit
                                      (permute (_ ,(BVN OBJ) ,RANGE)
                                               ,@(ITER (FOR OBJ BELOW OBJS)
                                                       (COLLECT (VPN VAR OBJ))))
                                      ,(VN VAR))
                                     ,(VN VAR)))))
                         (IF (ZEROP VAR)
                             CENTERED
                             `(bvand ,CENTERED
                                     ,(ALIGNED OBJ (1- VAR)))))))
              (LOOP :FOR OBJ :BELOW OBJS :COLLECT
                 `(= (_ bv0 ,(* 2 RANGE))
                     (bvxor ,(ALIGNED OBJ (1- VARS))
                            (_ ,(BVN (EXPT 2 (1- RANGE)))
                               ,(* 2 RANGE))))))))

        (check-sat)
        (get-model))))
```