Towards a Hybrid Approach to Protect Against Memory Safety Vulnerabilities

Kaled Alshmrany*, Ahmed Bhayat*, Lucas Cordeiro*, Konstantin Korovin*,
Tom Melham†, Mustafa A. Mustafa*, Pierre Olivier*, Giles Reger*, Fedor Shmarov*
*The University of Manchester, †University of Oxford

***Abstract*—Memory corruption bugs continue to plague low-level systems software generally written in unsafe programming languages. In order to detect and protect against such exploits, many pre- and post-deployment techniques exist. In this position paper, we propose and motivate the need for a *hybrid* approach for the protection against memory safety vulnerabilities, combining techniques that can identify the presence (and absence) of vulnerabilities pre-deployment with those that can detect and mitigate such vulnerabilities post-deployment. Our hybrid approach involves three layers: hardware runtime protection provided by capability hardware, software runtime protection provided by compiler instrumentation, and static analysis provided by bounded model checking and symbolic execution. The key aspect of the proposed hybrid approach is that the protection offered is greater than the sum of its parts – the expense of post-deployment runtime checks is reduced via information obtained during pre-deployment analysis. During pre-deployment analysis, static checking can be guided by runtime information.**

I. Introduction

Memory errors in low-level systems software written in unsafe programming languages such as C or C++ represent one of the main problems in computer security [1]. In particular, in the MITRE ranking [2], the top ten vulnerabilities include four types of memory errors. Microsoft reports that around 70% of all security updates in their products address memory issues [3], and Google reports a similar number regarding bugs in the Chrome Browser [4].

Techniques to detect memory errors can be broadly classified in two categories: detecting and removing vulnerabilities before deployment [5]–[8], or detecting and mitigating them post deployment [9]–[17]. Post-deployment techniques necessarily run as part of the executed code, i.e., at runtime. Pre-deployment techniques are more diverse, including runtime techniques designed to be used as part of testing and static techniques that directly analyze the source code. Runtime techniques are exact because they check a set of concrete behavior defined by a set of given inputs. Conversely, static techniques aim to check all possible program behaviors but necessarily approximate this due to a lack of context and the well-known state-explosion problem (i.e., scalability limitations). Compared to static techniques, runtime techniques can be more generally applicable, but they may still introduce unacceptable overhead for post-deployment.

The result is a set of techniques with varying coverage and performance profiles (summarised in Section II). As an example of the trade-offs, consider the following C program containing a stack-use-after-return vulnerability and a subobject-bounds vulnerability as well as a number of issues that can prevent certain techniques from applying.

Listing 1. Introductory code example

```
1  #define LEN 1000000
2  struct my_type {
3    int* array[LEN];
4    int num;
5  } var;
6  void fun(int index) {
7      int a = 13;
8      // violates subobject bound for index=LEN
9      var.array[index] = &a;
10 }
11 void fun2() {
12     int a = 15;
13 }
14 int main(int argc, char *argv[]) {
15     // large loop hard for bounded model checking
16     for(int i=0; i <= LEN; i++)
17       fun(i);
18     printf("%d\n", *var.array[LEN]);
19     // Vulnerability depends on complex
20     // constraint on input parameters
21     if(argv[0] = 2*argv[1])
22       fun2();
23     // potentially leaks fun2 local variable
24     printf("%d\n", *var.array[LEN]);
25     return 0;
26 }
```

No single protection mechanism detects both vulnerabilities. The stack-use-after-return vulnerability can be detected by the runtime SoftBound tool but this mechanism fails to detect the subobject buffer overflow vulnerability. Conversely, hardware protection (provided by CHERI's hardware capabilities PureCap model) can, in a given mode, detect the subobject buffer overflow vulnerability but not the stack use-after-return vulnerability. All runtime mechanisms require suitable inputs to be given to be used pre-deployment. Bounded model checking can detect both vulnerabilities but it struggles to cope with the large loop bound.

In this position paper, we present an experimental analysis (Section III) that demonstrates that these techniques (or at least some tools representing them) are complementary in the sense that no tool captures all vulnerabilities. We then propose a hybrid framework (Section IV) that aims to combine techniques but also, most interestingly, provides an opportunity for *cooperation*. Our goal is to combine techniques that (i) work with legacy code, (ii) do not require modification to the source code, and (iii) provide a low barrier to adoption. This goal guides our choice of memory protection techniques in this work.

II. Memory Protection Techniques

There are two main approaches to detecting memory errors *pre-deployment – runtime* techniques that aim to identify potential errors with a high overhead restricting them to pre-deployment test runs, as well as *static* techniques that explore the possible behaviors of the program without executing it. The main approach to providing protection *post-deployment* is to check memory accesses to ensure that they are safe – this may be via compiler-level instrumentation or via hardware support with new technologies that go beyond the traditional page table-based protection (e.g., Intel MPX [18], MPK [19], or hardware capabilities [17]). Post-deployment usually provides strong assurance against vulnerabilities but discovery of vulnerabilities (or false positives) at the post-deployment stage can lead to considerable disruptions. Below we outline the main techniques for runtime and static analysis.

A. Runtime Analysis

Checking memory access at runtime requires additional work. There is a trade-off between the amount of security provided and the level of overhead required. Often, techniques with large overheads are deemed incompatible with post-deployment except in the most security-critical settings.

Runtime checks may occur in the software or hardware. In software, such checks are typically inserted by the compiler. However, how this is performed and the overhead/coverage profile varies between tools. Alternatively, checks may be supported by unique hardware mechanisms. In this work, we initially consider three runtime analysis techniques:

- AddressSanitizer (ASAN) [6]: This tool uses a combination of shadow memory and so-called red zones with poisoning to detect spatial errors and a special memory allocator that provides address quarantine to detect temporal errors (with extra checks behind options). Developers suggest that $\sim$ 2x slowdown is standard.
- SoftBoundCETS[1] (SB) [9], [20]: This tool tracks pointers' metadata (*e.g.*, base, bound) using shadow space inspired mechanisms (instead of fat pointers) and uses this to insert checks into LLVM IR code to detect spatial and temporal errors. Experimental results [20] report average ~ 2.16x slowdown (up to 4x).
- PureCap [17]: The CHERI model implements memory access capabilities enforced by the hardware. A capability is a token giving access to a particular area of the virtual address space. In the PureCap model, each pointer of a C/C++ program is represented by a capability that carries metadata about the buffer bounds, access rights, etc. One of the advantages of PureCap is that it provides protection at the hardware level rather than intermediate levels that rely on correct implementation of compilers/machine code translation. Limitations include the need for specialised hardware and an increase in pointer sizes ($\sim 2x$) and corresponding increase in memory consumption.

A limitation of runtime techniques for pre-deployment checking is the need for concrete inputs. One method for addressing this is *fuzzing* [21], which attempts to find inputs that produce specific behaviors.

B. Static Analysis

Static techniques analyze the source code itself, searching the possible set of execution traces. There are, broadly, two main approaches: *breadth-first* bounded-model checking [22] unrolls the program, representing the reachability of a particular state by any path as a verification condition; and *depth-first* path-based symbolic execution [23] encodes a single path through the program as a set of symbolic constraints. Memory safety is cast as reachability of an unsafe state, and a satisfying assignment to the produced verification condition represents a *counter-example*, e.g., a set of inputs that leads to the error. In this work, we initially consider two static analysis tools since they achieved high positions in recent software verification (SV-COMP 2021 [24]) and software testing (Test-Comp 2021 [25]) competitions:

- ESBMC [8], [26]: This is a bounded-model checker utilizing Clang to transform C programs into an intermediate GOTO language. This is then symbolically executed, producing verification conditions for SMT solvers.
- FuSeBMC [27], [28]: This is a white-box fuzzer that injects labels into C programs and then use a combination of ESBMC and a path-based symbolic execution tool called Map2check [29] to find inputs that reach those labels (while checking for vulnerabilities).

III. Experimental Analysis

We perform an experimental analysis[2] with the selected tools using benchmarks taken from the 2021 memory safety category of SV-COMP [24], which contain various open-source applications, e.g., bftpd, which is an FTP server for Unix systems. We aim to demonstrate and explore their complementary nature. We begin by highlighting existing evidence; for example, the results of the most recent SV-COMP competitions [30] show that different techniques find different errors. We split our experimental analysis between benchmarks with given inputs and those without given inputs as the appropriate tools differ.

A. Programs with No Required Input

We run all tools on the 178 memory-safety benchmarks from SV-COMP 2021, where no input is required. We set the time limit of each run to 900 seconds (the SV-COMP time limit). These benchmarks are representative of a broad cross-section of essential vulnerabilities. They vary in size and complexity but are generally small, focusing on the vital vulnerability while being indicative of real-world scenarios.

The results are in Table I. The first thing to note is that every tool detects a different set of vulnerabilities. Runtime techniques detect more than static techniques, which is unsurprising as there is only a single behavior to analyze. However, static techniques detect some vulnerabilities, which runtime techniques fail to detect. One interesting case is a *potential*

[1]Available on GitHub, https://github.com/santoshn/softboundcets-34

[2]Scripts and data available at https://github.com/scorch-project/analysis.

TABLE I
SV-COMP21 BENCHMARKS WITH NO REQUIRED INPUTS.

Technique	Correct	Incorrect	Timeout
ASAN	159	13	6
SB	152	20	6
PureCap	145	24	9
ASAN + SB	166	6	6
ESBMC	130	5	43
FuSeBMC	133	4	41
Runtime (combined)	166	6	6
Static (combined)	132	5	41

stack-use-after-scope vulnerability that is not triggered in the program but presents a future vulnerability detected by static techniques but not by runtime techniques.

Combining all three runtime tools (by taking the maximum set of reported bugs) produces six incorrect verdicts. The interesting cases are false negatives (existing bugs not reported) due to ASAN failing to detect invalid memory cleanup (SB and PureCap do not handle memory leaks) and a false positive (from SoftBoundCETS) falsely reports a bug due to a lack of support for the C library function `memcpy`.

On the other hand, ASAN detects nine bugs that SB and PureCap do not detect, and for SB this number is 6. In contrast, PureCap did not detect any unique vulnerabilities (but should ultimately have a better performance profile).

In terms of performance, the current PureCap implementation used in the analysis is a prototype software model (emulated capability hardware) that does not give realistic performance numbers. Therefore, we compare the runtime overhead of ASAN and SB. The mean overhead for ASAN was 4.10x and for SB it was 4.46x but there was significant variance - 27.91 for ASAN and 96.23 for SB. We note that the amount of overhead introduced in safe benchmarks is significantly lower (2.33x± 0.28 for ASAN and $1.01x \pm 0.04$ for SB) than the unsafe ones (7.54x± 64.4 and 12.27x± 226.29). This is due to the relatively short runtime of the evaluated benchmarks ($0.11s \pm 0.23s$ and $0.14s \pm 0.25s$) in comparison to the overhead introduced by the termination procedure after finding a vulnerability.

The static techniques demonstrated significantly more timeouts even though each program had a single path. In 5 cases, ESBMC produced incorrect answers: in one case, it could not detect a comparison of freed pointers, and in the remaining four, it reported a bug in a safe code (it wrongly identified freeing the memory twice in case of using structures featuring bit fields; the code example replicating this bug can be found in Listing 11). FuSeBMC repeated 4 (including the comparison of freed pointers) out of these five incorrect verdicts. As a result, combination of both tools yields one more incorrect outcome than FuSeBMC alone as ESBMC returns an incorrect "unsafe" verdict for a safe benchmark for which FuSeBMC provides a correct outcome.

B. Programs Requiring Input

We run ESBMC and FuSeBMC on the 127 unsafe benchmarks from SV-COMP 2021, where input is required for 900 seconds. We do not run Map2Check directly as it performed very poorly outside of the FuSeBMC setup. The results are in Table II. Both FuSeBMC and ESBMC returned incorrect verdicts for two benchmarks (undetected memory leaks). The bugs were not detected because ESBMC and FuSeBMC do not provide an implementation of a C library function `atexit` (Listing 10 reproduces this bug). At the same time, ESBMC reached the timeout in 8 more cases (17 vs 9). For the unsafe verdicts, both ESBMC and FuSeBMC produced counter-examples (*i.e.*, inputs) violating memory safety. Such inputs can be introduced into the original code (and possibly combined with the described runtime verification techniques) for further testing.

TABLE II
SV-COMP21 BENCHMARKS WITH INPUTS.

Technique	Correct	Incorrect	Timeout
ESBMC	107	2(1)	17
FuSeBMC	116	2	9
Combined	116	2	9

C. Vulnerability Analysis

We have identified vulnerabilities that cannot be detected by at least one of the selected tools during experiments and our exploration. These are summarized in Table III and briefly discussed below.

a) Subobject-buffer-overflow: ASAN and SB do not track subobject bounds, so do not detect these vulnerabilities. PureCap has an additional option (requiring extra checks) that can detect subobject bounds. However, in some cases, this leads to more false positives, e.g., when performing pointer arithmetic on a pointer to a subobject [31].

b) Use-after-free: PureCap cannot detect this vulnerability as the current stable release only supports spatial safety. There is an experimental release based on CHERIvoke [32], which quarantines freed memory, but (for specific performance reasons) this does not handle use-after-free, rather the more specific use-after-reallocate vulnerability.

c) Stack-use-after-return: PureCap explicitly does not handle stack exploits, which would require complex (and expensive) revocation mechanisms. ASAN does not support this by default, although some versions (not the one we used) provide an option for additional checks. (In order to enable this, one needs to set an environment variable `ASAN_OPTIONS=detect_stack_use_after_return=1`.)

d) Stack-use-after-scope: PureCap cannot handle these stack-based vulnerabilities. SB cannot detect this as the scoping information is not handled during its instrumentation phase at the intermediate level of the LLVM compiler.

e) Double-free: This is an example of a temporal memory safety vulnerability that the Cornucopia [33] extension of PureCap could detect, but the stable version does not.

f) Memory-leaks: SBC and PureCap do not explicitly track memory and cannot detect this class of vulnerability.

g) Unions: PureCap does not support some program features. For example, due to separating pointers from other data and the larger pointer sizes, PureCap can incorrectly report buffer-overflow when unions are used.

TABLE III
FEATURES SUPPORTED BY DIFFERENT PROTECTION METHODS.

Feature	ASAN	SB	PureCap (RISC-V)	ESBMC	FuSeBMC
Spatial Memory Safety					
Subobject buffer overflow	**no**	**no**	**no**/yes	yes	yes
Temporal Memory Safety					
Use-after-free	yes	yes	**no**	yes	yes
Stack use after return	**no**/yes	yes	**no**	yes	yes
Stack use after scope	yes	**no**	**no**	yes	yes
Double free	yes	yes	**no**	yes	yes
Memory leaks	yes	**no**	**no**	yes	yes
Program Features					
Unions	yes	yes	yes/**no**	yes	yes
Library functions	yes/no	yes/no	yes/no	yes/no	yes/no

h) Library Functions: It is worth noting that all mechanisms require access to the source code of any library functions in some way. SB and ESBMC provide mechanisms that allow the behavior of library calls to be emulated. SB, ASAN, and PureCap require external code to be compiled with the appropriate checks to provide coverage (and PureCap requires compatibility due to the different pointer sizes). ESBMC will over-approximate the behavior of library calls, but this can lead to many spurious false positives.

D. Examples of Vulnerabilities

In this section we present short code examples of memory safety vulnerabilities outlined in Table III.

Listing 2. Subobject buffer overflow

```
1  #define LEN 1
2  struct my_type {
3      int array[LEN];
4      int num;
5  } var;
6
7  int main() {
8      var.array[LEN] = 0;
9      return 0;
10 }
```

The runtime protection techniques do no find any bugs in the code in Listing 2 where the array in the structure `var` is assigned a value beyond its value, while the bound of the object `var` is not violated. However, this instance of subobject bound violation is identified by ESBMC and CBMC. (When line 4 is commented out SoftBoundCETS and PureCap find the error, as this results in the global buffer overflow.)

Listing 3. Use after free

```
1  int main() {
2      int *i;
3      i = (int *)malloc(sizeof(int));
4      free(i);
5      *i = 15;
6      return 0;
7  }
```

Listing 4. Stack use after return

```
1  int *i;
2  void fun() {
3      int b = 13;
4      i = &b;
5  }
6  void fun2()  {
7      int a = 15;
8  }
9  int main()  {
10     fun();
11     printf("%d\n", *i);
12     fun2();
13     printf("%d\n", *i);
14     return 0;
15 }
```

In this example, pointer `i` can potentially gain access to the local variable `a` of function `fun2()` after it has returned. For example, when this code is compiled with Clang without any optimisation options and executed, the second `printf` will output `15` (on both capability-enabled and regular platforms). This can cause a safety problem if variable `a` contains some sensitive information (e.g., a secret key). This bug is detected by SoftBoundCETS and both bounded model checkers (ESBMC and CBMC). At the same time AddressSanitizer does not report this violation by default, however the second `printf` outputs the value `13` instead of `15`. Nevertheless, this bug is detected when the environment variable `ASAN_OPTIONS` is assigned with the value detect_stack_use_after_return=1.

Listing 5. Stack use after scope

```
1  int main() {
2      int *ptr; {
3          int var = 0;
4          ptr = &var;
5      }
6      *ptr = 1;
7      return 0;
8  }
```

SoftBoundCETS and PureCap do not protect against stack use after scope (see the example above).

Listing 6 contains a potential memory safety issue because `p` points to the address of a local (with the respect to `p`) array declared in the `for` loop. This code is supposed to be unsafe according to the SV-COMP expected outcome, and ESBMC returns `dereference failure: accessed expired variable pointer a`. However, AddressSanitizer does not find any issues with this code because despite having an address of `a`, pointer `p`

is never dereferenced after the loop. And as soon as `p` is dereferenced, AddressSanitizer detects the error.

Listing 6. Stack use after scope 2

```
1  int main() {
2      int* p;
3      for(int i = 0; i < 2; i++)  {
4          int a[10];
5          if(i == 0)
6              p = a;
7          else
8              p[0] = 1;
9      }
10     return 0;
11 }
```

Listing 7. Double free

```
1  int main() {
2      int *i;
3      i = (int *)malloc(sizeof(int));
4      free(i);
5      free(i);
6      return 0;
7  }
```

Listing 8. Memory leak

```
1  int main() {
2      int *i;
3      i = (int*)malloc(sizeof(int));
4      return 0;
5  }
```

Listing 9. Union misuse

```
1  #define SIZE 8
2  int main() {
3      union {
4          int *p0;
5          struct {
6              char p1[SIZE];
7              int p2;
8          } str;
9      } data;
10     data.p0 = (int*) malloc(sizeof(int));
11     data.str.p2 = 1;
12     free(data.p0);
13     return 0;
14 }
```

Unions pose compatibility issues for a capability-enabled architecture. This is because memory is allocated for the largest member of the union, and all the members of the union share this memory. Thus, in the example above pointer `data.p0` (8 bytes) and variable `data.str.p2` (4 bytes) will be separated in memory (as `data.str.p2` is "padded" by 8 bytes of the array `data.str.p1`) on a platform not supporting capabilities. At the same time this code will result into an invalid capability `data.p0` on a capability-enabled platform as variable `data.str.p2` will overwrite 4 bytes of `data.p0` (16 bytes).

Listing 10. Unsupported `atexit` function

```
1  #include <stdlib.h>
2  void *ptr;
3  void free_memory(){
4    free(ptr);
5  }
6  int main(){
7    ptr = malloc(1);
8    atexit(free_memory);
9    return 0;
10 }
```

ESBMC and FuSeBMC report a memory leak in the code above. However, it can be seen that this example is memory leak free as memory cleanup happens upon reaching the return of `main` via calling `free_memory` which is registered to be executed upon the program exit using a C library function `atexit`. This function is not yet supported by ESBMC and FuSeBMC which is the reason of the incorrect verdict.

Listing 11. A false positive featuring bit fields

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdio.h>
4  struct str {
5      unsigned char a:1;
6      unsigned char b:1;
7      unsigned char c;
8  } var;
9  int main() {
10     struct str *ptr;
11     ptr = malloc(2);
12     var.a = 0;
13     var.b = 0;
14     var.c = 255;
15     memcpy(ptr, &var, 2);
16     if (ptr->a) {
17         free(ptr);
18     }
19     if (ptr->b) {
20         free(ptr);
21     }
22     if (ptr->c != 255) {
23         free(ptr);
24     }
25     free(ptr);
26     return 0;
27 }
```

In the code above ESBMC and FuSeBMC wrongly identify a "double free" of pointer `ptr` at lines 20 and 25. Perhaps this could be caused by bugs in the implementation of `memcpy` in ESBMC and FuSeBMC.

E. Summary

Our experimental analysis supports the motivation that runtime and static techniques can complement each other for pre- and post-deployment protection. Interestingly, PureCap provides a subset of safety guarantees that are expected to be very cheap, suggesting a hybrid setup where PureCap handles these cheap checks. In contrast, the rest are handled by in-software checks – this is what we propose next.

IV. Proposed Hybrid Framework

Our proposed hybrid framework is illustrated in Fig. 1. This combines static and runtime protection mechanisms to offer protection at both pre- and post-deployment stages. Whilst

combining techniques is not a new idea (e.g. [34]), our focus is on the combination across different deployment stages.

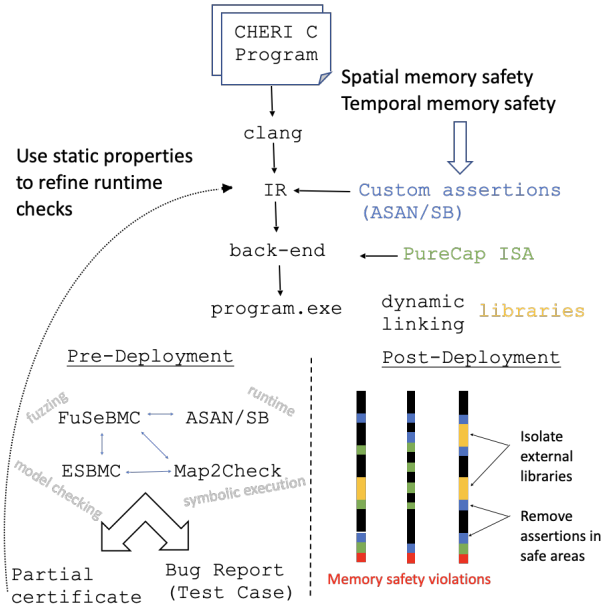

Fig. 1. Proposed Hybrid Framework.

The framework utilizes the LLVM toolchain for (i) the insertion of assertions, (ii) the translation of C code for static analyzers, and (iii) the compilation to PureCap ISA. Conveniently, the selected tools already use this toolchain. The goal is to provide an *architecturally independent* set of memory safety guarantees with a minimal performance impact. Therefore, to maximize our framework's adoption, compilation to PureCap will be optional, with runtime checks performed by compiler-inserted assertions for non-capability hardware.

By combining techniques, we aim to provide the union of protection coverage as 'cheaply' as possible, selecting the cheapest way to provide each check (noting that some methods are incompatible with some compiler optimizations). Below we outline the main directions where cooperation of different techniques can lead to the development of such hybrid framework.

A. Isolating Libraries

As previously discussed, a problematic issue for all techniques is the interaction with external libraries. We assume various methods to compartmentalize the program and isolate the protected code from external libraries that are not subject to memory safety protection. Hardware memory capabilities [35] are one of the most efficient technologies to achieve that, providing exception-less security domain transitions and efficient cross-compartment communication through capabilities.

Many other compartmentalization abstractions can be used for platforms that do not support hardware capabilities, relying on various isolation mechanisms. These can be process-based isolation leveraging page tables [36], [37]; VM-based isolation using hardware-assisted virtualization [38], [39]; trusted execution environments [40], [41] and other ISA extensions such as Intel MPK [42]–[45]; and finally software-only solutions such as SFI [46]. These techniques offer various security/performance trade-offs and generally require a particular porting effort to manage data shared between compartments.

B. Certifying the Removal of Assertions

As well as detecting bugs, static tools can certify the absence of specific bugs in some or all of the code to achieve partial or complete certification. Here, k-induction [47], [48] can be used to prove a safety property ϕ for any given depth of the program's state space. The main idea is to use an iterative deepening approach and check, for each step k up to a maximum value, that ϕ holds with in all states reachable within k iterations and that if ϕ holds for k iterations, it holds for the subsequent unfolding of the system. The main challenge of this approach relies on computing and strengthening loop invariants, which must be inductive (and not just invariant) to check the corresponding verification conditions [49]. Such *certificates* will be used to identify runtime checks that are no longer necessary and can be removed. We will also explore the leverage of (cheap) assurances from PureCap in this process i.e. explore whether (software-based) runtime checks can be removed when assuming the protection offered by PureCap.

C. Safe under Assumptions

Combining the two previous ideas and isolating unknown code, we will also explore the isolation of safe code, e.g., where some safe code is statically shown safe under certain assumptions (typically at entry) or invariants, we will insert runtime checks to check those assumptions or invariants. We may also be able to prove safety under *additional* assumptions, e.g., replace a series of expensive runtime checks with fewer, cheaper ones. Finally, information about isolation can be used within the static analysis to modularise the checking process to (partially) address the state-explosion issue.

D. Static Analysis to Support Capability Revocation

One of the main limitations of capability-based hardware within the context of temporal memory safety is the need to *revoke* permissions and the overhead this requires. We propose using static analysis methods to identify when capabilities should be revoked and insert these directly into the code. For example, this should increase the number of *use-after-free* bugs detectable by the CHERIvoke [32] extension of PureCap.

V. Conclusion

This paper motivates and describes a proposed hybrid framework for memory safety protection. We analyze some techniques and tools for providing memory safety protection and identify areas in which they complement. We then propose a hybrid framework that aims to achieve joint coverage as cheaply as possible. Finally, we identify further research

directions to take advantage of the potential cooperation of the combined techniques.

ACKNOWLEDGEMENT

This work was undertaken as part of the *SCorCH: Secure Code for Capability Hardware* project funded by EPSRC and Innovate UK as part of the Digital Security by Design (DSbD) challenge.

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[2] MITRE, "Mitre's top 25 cwe," 2020, https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[3] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues," 2019, https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[4] Google, "https://www.chromium.org/home/chromium-security/memory-safety," 2020, https://www.chromium.org/Home/chromium-security/memory-safety.

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Annual Tech. Conf. (ATC'12)*, 2012, pp. 309–318.

[7] T. Kremenek, "Finding software bugs with the clang static analyzer," *Apple Inc*, 2008.

[8] L. C. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[9] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, vol. 44, no. 6, p. 245–258, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542504

[10] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *In 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 128–139.

[11] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*. Citeseer, 2003, pp. 243–255.

[12] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Tran. on Inf. and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[14] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 147–160.

[15] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 573–584.

[16] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2389–2403.

[17] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *ACM/IEEE 41st Int. Symposium on Computer Architecture (ISCA)*, 2014, pp. 457–468.

[18] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018.

[19] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel {MPK})," in *2019 USENIX Annual Tech. Conf. (USENIX ATC 19)*, 2019, pp. 241–254.

[20] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," *SIGPLAN Not.*, vol. 45, no. 8, p. 31–40, Jun. 2010. [Online]. Available: https://doi.org/10.1145/1837855.1806657

[21] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[22] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, And Tools*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2006.

[23] J. C. King, "Symbolic Execution And Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[24] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *27th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, vol. 12652, 2021, pp. 401–422.

[25] ——, "Status report on software testing: Test-comp 2021," in *24th International Conference Fundamental Approaches to Software Engineering (FASE)*, ser. LNCS, vol. 12649, 2021, pp. 341–357.

[26] M. Y. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: an industrial-strength C model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering ASE*. ACM, 2018, pp. 888–891.

[27] K. M. Alshmrany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, "FuSeBMC: A white-box fuzzer for finding security vulnerabilities in C programs (competition contribution)," in *24th Int. Conf. Fundamental Approaches to Software Engineering (FASE)*, ser. LNCS, vol. 12649, 2021, pp. 363–367.

[28] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs," *TAP*, 2021, awaiting Publication.

[29] H. Rocha, R. Menezes, L. C. Cordeiro, and R. S. Barreto, "Map2check: Using symbolic execution and fuzzing - (competition contribution)," in *26th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 12079, 2020, pp. 403–407.

[30] D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 12652, p. 401, 2021.

[31] B. D. J. B. D. C. J. C. N. F. S. W. M. E. N. P. S. Robert N. M. Watson, Alexander Richardson and P. G. Neumann, ""cheri c/c++ programming guide, technical report ucam-cl-tr-947"," Computer Laboratory, Cambridge, Tech. Rep., June 2020.

[32] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson *et al.*, ""cherivoke": Characterising pointer revocation using cheri capabilities for temporal memory safety," in *52nd Annual IEEE/ACM Int. Symposium on Microarchitecture*, 2019, pp. 545–557.

[33] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin *et al.*, "Cornucopia: Temporal safety for cheri heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 608–625.

[34] D. Beyer and H. Wehrheim, "Verification artifacts in cooperative verification: Survey and unifying component framework," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2020, pp. 143–167.

[35] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE Symp. on Security and Privacy*, 2015, pp. 20–37.

[36] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix." in *USENIX Security Symposium*, vol. 46, 2010, p. 2.

[37] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 219–232.

[38] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 116–132.

[39] H. Lefeuvre, V.-A. Badoiu, S. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, and C. Raiciu, "Flexos: Making os isolation flexible," in *The 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*, 2021.

[40] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[41] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, pp. 1–36, 2019.

[42] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel {MPK})," in *2019 USENIX Annual Tech. Conf. (USENIX ATC 19)*, 2019, pp. 241–254.

[43] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "{ERIM}: Secure, efficient in-process isolation with protection keys ({MPK})," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[44] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *USENIX Annual Tech. Conf.*, 2019, pp. 489–504.

[45] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *16th ACM SIGPLAN Int. Conf. on Virtual Execution Environments*, 2020, pp. 143–156.

[46] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[47] M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of C programs via k-induction," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 1, pp. 97–114, 2017.

[48] M. Y. R. Gadelha, F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole, "ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 11429, 2019, pp. 209–213.

[49] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.