Position Paper: Towards a Hybrid Approach to Protect Against Memory Safety Vulnerabilities

Kaled Alshmrany*, Ahmed Bhayat*, Franz Brauße*, 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 proposed 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 potentially 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 and can make software more susceptible to denial of service attacks due to preventive program termination even in the cases of non-critical bugs.

The result is a set of techniques with varying coverage and performance profiles (summarized in Section II). Listing 1 gives a C program demonstrating the trade-offs. It contains a *stack-use-after-return* vulnerability (lines 13 and 26) and a *subobject-bounds* vulnerability (line 6). The code contains features that expose limitations of existing analysis techniques, as described next.

Listing 1. Code example demonstrating trade-offs between techniques.

```
1  #define LEN 1000000
2  struct my_type {int* array[LEN]; int* num;} var;
3  void fun(int index) {
4    int a = 13;
5    // overflows subobject bound for index=LEN
6    var.array[index] = &a;
7  }
8  void fun2() { int unused_var; int a = 15; }
9  int main(int argc, char *argv[]) {
10   // large loop hard for bounded model checking
11   for(int i=0; i <= LEN; i++) fun(i);
12   // accesses expired local variable 'a' of 'fun'
13   printf("%d\n", *var.array[0]);
14   int a = atoi(argv[1]), b = atoi(argv[2]);
15   int c = atoi(argv[3]), d = atoi(argv[4]);
16   // function call to 'fun2' depends on
17   // complex constraint on program's inputs
18   if(a == 2*b + c*d) fun2();
19   // depending on the compiler design of
20   // the call stack, this may leak
21   // local variable 'a' of 'fun2'
22   // as it may be placed at the same address
23   // as variable 'a' in line 4
24   // (e.g., happens when compiled with Clang
25   // without optimisation flags or -O0)
26   printf("%d\n", *var.array[0]);
27   return 0;
28 }
```

No single protection mechanism detects both vulnerabilities. The *stack-use-after-return* vulnerability in Listing 1 can be detected by the runtime software-level protection tools (e.g., SoftBoundCETS [9], [18] and AddressSanitizer [6]; see below for details), but these mechanisms fail to detect the subobject buffer overflow vulnerability. Conversely, hardware protection (provided by CHERI's hardware capabilities PureCap model [17], [19] as described below) is capable of detecting the subobject buffer overflow vulnerability but not the *stack-use-after-return* vulnerability. Any runtime mechanisms requires suitable inputs (lines 14 and 15) to be given to be used during pre-deployment, and its ability to detect errors depends directly on those inputs (line 18). The static bounded model checking approach (as implemented in the ESBMC tool [8], [20] described below) can detect both vulnerabilities, but it struggles to cope with the large loop bound (line 11).

The above motivates the development of a hybrid framework that can provide consistent software security at an optimal (i.e., controlled) performance cost. Our focus is on combining techniques across different deployment stages. In particular, we are interested in leveraging capability hardware to provide post-deployment security without a major performance sacri-

fice. Combining techniques is of course not a new idea. For example, the Frama-C framework is built on the concept of combining a variety of techniques, including pre- and post-deployment [21]. However, in contrast to our proposed framework, Frama-C is not fully automatic and does not make use of runtime hardware protection. Furthermore, we provide novel ideas for a *fine grained* combination of static and runtime techniques (Sections IV-B, IV-C and IV-D), that we think will reduce the penalty associated with achieving security. Other attempts at combining techniques to achieve greater program safety include [22]–[24]. However, none of these combine the full range of pre- and post-deployment methods that we propose, nor do they utilise our ideas relating to the manner of combining these techniques. We are proposing a paradigm shift from the "all or nothing" in terms of the method of software protection strategy to a flexible hybrid verification approach, combining hardware safety guarantees with optimisable software hardening.

Consequently, the proposed hybrid framework must incorporate three essential cooperating components:

1) Capability hardware that efficiently protects against a subset of memory vulnerabilities at runtime in hardware.
2) Ahead-of-compilation formal verification tooling capable of proving the program's safety in certain cases [1].
3) A software hardening engine that can instrument the program with the software safety checks that a) are not yet provided by the capability hardware and b) are not proved to be "safe" by the formal verification tool.

Therefore, the proposed hybrid framework covers both software protection modes: pre-deployment and post-deployment.

A typical trade-off is that between security and cost. The high cost of software hardening has often been prohibitive. This framework addresses this by: (i) leveraging new hardware protection to move most expensive software checks to hardware; (ii) utilizing static analysis tools to prove that certain software checks are not required; and (iii) enabling the complementary application of automated testing techniques (e.g., fuzzing) during pre-deployment to increase the likelihood that bugs are resolved at this more cost-effective stage. All three points are achieved without compromising the overall security guarantees.

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 catches all vulnerabilities that at least one tool can catch. 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 (and tools) in this work. Currently we are implementing the proposed hybrid framework (see Section IV for current implementation status).

[1]Tools based on Bounded Model Checking (BMC) are capable of formally proving safety of bounded executions of the given program (i.e., "partial safety"). Also it is often possible to explore the entire state space of the given program (i.e., when all possible executions of the given program are bounded) and produce a formal proof of program's safety.

II. Memory Protection Techniques

There are two main approaches to detecting memory errors *pre-deployment – runtime* techniques that aim to identify potential errors but 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 [25], MPK [26], 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 consider three runtime analysis tools that are some of the most widely used runtime protection tools utilising the Clang/LLVM toolchain which we consider to be the integral part of our hybrid framework introduced in Section IV:

- 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[2] (SB) [9], [18]: 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 [18] report average $\sim$ 2.16x slowdown (up to 4x).
- The pure capability model of CHERI (PureCap) : The CHERI model [17] 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 of CHERI, each pointer of a C/C++ program is represented by a capability that carries metadata about the buffer bounds, access rights, etc. CHERI also implements a hybrid model [19] that allows the coexistence of regular pointers (without any hardware-level protection guarantees) and capabilities that have to be declared explicitly. One of the advantages of PureCap is that it provides protection at the hardware level rather than intermediate levels that rely on correct

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

implementation of compilers/machine code translation. Limitations include the need for specialized 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* [27], 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 [28] unrolls the program, representing the reachability of a particular state by any path as a verification condition; and *depth-first* path-based symbolic execution [29] 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 consider two static analysis tools since they have been achieving high positions in recent software verification (SV-COMP [30]) and software testing (Test-Comp [31]) competitions, and they both utilise the Clang/LLVM toolchain (this is essential for the hybrid framework we propose in Section IV):

- ESBMC [8], [20]: 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 [32], [33]: injects labels into C programs for tracking code coverage and uses a combination of white-box fuzzing and ESBMC to find inputs that reach those labels (while also checking for vulnerabilities).

III. Experimental Analysis

To establish the complementary nature of existing tools, we performed an experimental analysis[3] with the selected tools using benchmarks taken from the memory safety category of SV-COMP [34], which contain various open-source applications, e.g., *bftpd*, which is an FTP server for Unix systems. Given our aim of establishing that tools are complementary, let us highlight existing evidence from the most recent SV-COMP competitions [35] which immediately shows that different techniques find different errors. Our experimental analysis is divided into two parts: 1) quantitative, where we determine how many vulnerabilities can be detected by the compared tools (see Tables I and II), and 2) qualitative, where we establish the types of vulnerabilities that can or cannot be handled by the given tool (see Table III). We further split our quantitative analysis between benchmarks with given inputs (Section III-A) and those without given inputs (Section III-B) as the appropriate tools differ. For benchmarks that require inputs, it is difficult to accurately compare static and runtime tools, since for runtime tools to be able to detect an error, they would require the precise inputs that trigger the error.

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

TABLE I
SV-COMP benchmarks with no required inputs. FN stands for false negative - where a tool fails to identify a genuine bug. FP stands for false positive.

Technique	Correct	Incorrect (FN+FP)	Timeout
ASAN	159	13 (13+0)	6
SB	152	20 (19+1)	6
PureCap	145	24 (24+0)	9
ASAN + SB	166	6 (5+1)	6
Runtime (combined)	166	6 (5+1)	6
ESBMC	130	5 (1+4)	43
FuSeBMC	133	4 (1+3)	41
Static (combined)	132	5 (1+4)	41

A. Programs with No Required Input

We run all tools on the 178 memory-safety benchmarks from SV-COMP, where no input is required. We set the time limit of each run to 900 seconds (the time limit used at SV-COMP). 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 *combined* rows represent the results that could be achieved by running the tools as a portfolio and taking the output to be "bug" if any tool returns "bug", "safe" if no tool returns "bug" and at least one tool returns "safe", and timeout otherwise. Note 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 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 (in the manner described above) produces six incorrect verdicts. The interesting cases are the false negatives due to ASAN failing to detect invalid memory cleanup (SB and PureCap do not handle memory leaks) and a false positive from SoftBoundCETS falsely reporting 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 (emulating 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 ($2.33\text{x}\pm0.28$ for ASAN and $1.01x \pm 0.04$ for SB) than the unsafe ones ($7.54\text{x}\pm64.4$ and $12.27\text{x}\pm226.29$). 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

TABLE II
SV-COMP BENCHMARKS WITH INPUTS.

Technique	Correct	Incorrect (FN+FP)	Timeout
ESBMC	107	3 (3+0)	17
FuSeBMC	116	2 (2+0)	9
Combined	116	2	9

finding a vulnerability.

The static techniques demonstrated significantly more time-outs even though each program had a single path. Both tools were able to produce the correct verification outcomes in the majority of the cases. However, they produced incorrect verdicts in around 3% of benchmarks. In one of the cases both ESBMC and FuSeBMC could not detect a comparison of freed pointers. In 5 cases, ESBMC produced incorrect answers: in the case mentioned above as well as in four others where it reported a bug in a safe code due to wrongly identifying freeing memory twice in case of using structures featuring bit fields. FuSeBMC repeated 4 (including the comparison of freed pointers) out of these five incorrect verdicts. As a result, the combination of both tools, yields one more incorrect outcome (i.e., five overall) than FuSeBMC alone as ESBMC returns an incorrect "unsafe" verdict for a safe benchmark for which FuSeBMC provides a correct outcome. This is a general limitation of the ahead-of-compilation verification tools since they need to assume (i.e., approximate) the behaviour of the underlying hardware.

B. Programs Requiring Input

We run ESBMC and FuSeBMC on the 127 unsafe benchmarks from the SV-COMP memory safety category, where input is required (using the same timeout of 900 seconds for each run). 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`. Also, ESBMC crashed on one of the benchmarks. At the same time, ESBMC reached the timeout in 8 more cases (17 vs 9). (ESBMC does not produce any unique outcomes, while FuSeBMC detects 9 unique vulnerabilities thanks to its white-box fuzzer.) 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.

C. Vulnerability Analysis

We have identified issues, posing various degree of security risk, that cannot be detected by at least one of the tools used during experiments. Also we identified program features that can be problematic for cross-platform compatibility. 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 [36].

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 [37], which quarantines freed memory, but (for 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, since it may significantly slow down[4] the resulting executable. (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 [38] extension of PureCap could detect, but the stable version does not.

f) Memory-leaks: SB 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.

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. 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.

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.

We aim to provide the union of protection coverage as 'cheaply' as possible. Here cost can mean two things. Firstly, the framework should be as portable as possible to minimise development and deployment cost. Therefore, the goal is to

[4]https://docs.microsoft.com/en-us/cpp/sanitizers/error-stack-use-after-return?view=msvc-170

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

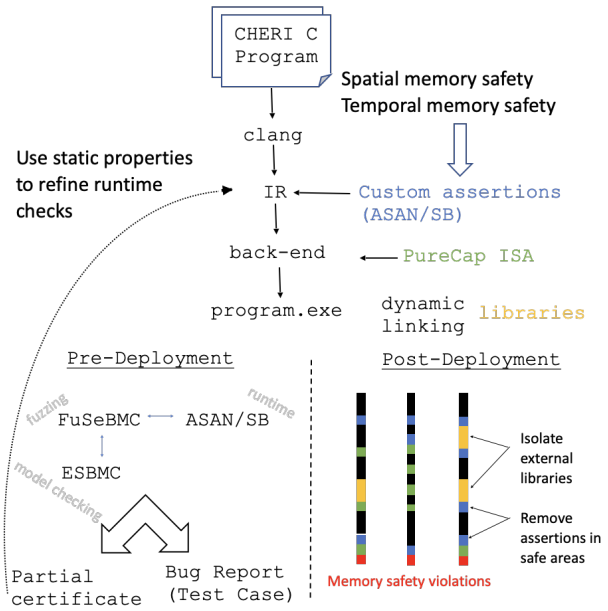

Fig. 1. Proposed Hybrid Framework.

provide an *architecturally independent* framework, allowing the same guarantees to be achieved across platforms. Thus, compilation to PureCap will be optional, with runtime checks performed by compiler-inserted assertions for non-capability hardware. Secondly, the runtime performance impact should be minimal. By combining techniques, we can select 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. We have already made progress towards implementing this framework by extending ESBMC with the support for CHERI capability hardware [39]. This extension verifies programs for different target hardware, utilising capabilities when the hardware enables them and keeping the same safety guarantees even when there are no dedicated hardware safety mechanisms.

A. Isolating Libraries

A problematic issue for all techniques is the interaction with external libraries which may (i) not have their own runtime protection, or (ii) be available for static analysis. 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 [19] 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 [40], [41]; VM-based isolation using hardware-assisted virtualization [42], [43]; trusted execution environments [44], [45] and other ISA extensions such as Intel MPK [46]–[49]; and finally software-only solutions such as SFI [50]. 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 [51], [52] 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 [53]. Such *complete and partial* certificates will be used to identify runtime checks that can be further refined (e.g., simplified) or removed (if they are proved to be no longer necessary). We will also explore leveraging (cheap) assurances from PureCap in this process by exploring whether (software-based) runtime checks (i.e., assertions) can be removed when assuming the protection offered by PureCap (during both pre- and post-deployment stages).

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 [37] 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.

References

[1] L. Szekeres *et al.*, "Sok: Eternal war in memory," in *IEEE SP'13*. IEEE, 2013, pp. 48–62.

[2] MITRE, "Mitre's top 25 cwe," 2022, https://cwe.mitre.org/top25/archive/2022/2022_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, 2022, https://www.chromium.org/Home/chromium-security/memory-safety.

[5] N. Nethercote *et al.*, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[6] K. Serebryany *et al.*, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC'12*, 2012, pp. 309–318.

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

[8] L. C. Cordeiro *et al.*, "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 *et al.*, "Softbound: Highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, vol. 44, no. 6, p. 245–258, 2009.

[10] G. C. Necula *et al.*, "Ccured: Type-safe retrofitting of legacy code," in *ACM POPL'02*, 2002, pp. 128–139.

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

[12] N. Burow *et al.*, "Sok: Shining light on shadow stacks," in *2019 IEEE SP*. IEEE, 2019, pp. 985–999.

[13] M. Abadi *et al.*, "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 *et al.*, "Securing software by enforcing data-flow integrity," in *OSDI'06*, 2006, pp. 147–160.

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

[16] S. Silvestro *et al.*, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC*, 2017, pp. 2389–2403.

[17] J. Woodruff *et al.*, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE ISCA'14*, 2014, pp. 457–468.

[18] S. Nagarakatte *et al.*, "Cets: Compiler enforced temporal safety for c," *SIGPLAN Not.*, vol. 45, no. 8, p. 31–40, Jun. 2010.

[19] R. N. Watson *et al.*, "Cheri: a hybrid capability-system architecture for scalable software compartmentalization," in *IEEE SP*, 2015, pp. 20–37.

[20] M. Y. R. Gadelha *et al.*, "ESBMC 5.0: an industrial-strength C model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 888–891.

[21] D. Pariente *et al.*, "Static analysis and runtime-assertion checking : Contribution to security counter-measures," 2017.

[22] D. Beyer *et al.*, "Verification artifacts in cooperative verification: Survey and unifying component framework," in *ISoLA'20*. Springer, 2020, pp. 143–167.

[23] M. Christakis *et al.*, "Guiding dynamic symbolic execution toward unverified program executions," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 144–155.

[24] S. Cruanes *et al.*, "Tool integration with the evidential tool bus," in *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 275–294.

[25] O. Oleksenko *et al.*, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *ACM POMACS'18*, vol. 2, no. 2, pp. 1–30, 2018.

[26] S. Park *et al.*, "libmpk: Software abstraction for intel memory protection keys (intel mpk)," in *USENIX ATC'19*, 2019, pp. 241–254.

[27] M. Sutton *et al.*, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[28] A. V. Aho *et al.*, *Compilers: Principles, Techniques, And Tools*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2006.

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

[30] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *TACAS'21*, vol. 12652, 2021, pp. 401–422.

[31] ——, "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.

[32] K. M. Alshmrany *et al.*, "FuSeBMC: A white-box fuzzer for finding security vulnerabilities in C programs (competition contribution)," in *FASE'21*, ser. LNCS, vol. 12649, 2021, pp. 363–367.

[33] ——, "FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs," p. 85–105, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-79379-1_6

[34] D. Beyer, "Progress on software verification: SV-COMP 2022," in *TACAS'22*, ser. LNCS 13244. Springer, 2022, pp. 375–402.

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

[36] R. N. M. Watson *et al.*, "CHERI C/C++ programming guide," Computer Laboratory, Cambridge, Tech. Rep., 2020.

[37] H. Xia *et al.*, "CHERIvoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *MICRO*, 2019, pp. 545–557.

[38] N. W. Filardo *et al.*, "Cornucopia: temporal safety for CHERI heaps," in *IEEE SP'20*. IEEE, 2020, pp. 608–625.

[39] F. Brauße *et al.*, "Esbmc-cheri: towards verification of C programs for cheri platforms with esbmc," in *ISSTA*. ACM, 2022, pp. 773–776.

[40] R. N. Watson *et al.*, "Capsicum: Practical capabilities for unix." in *USENIX Security'10*, vol. 46, 2010, p. 2.

[41] C. Reis *et al.*, "Isolating web programs in modern browser architectures," in *ACM EuroSys'09*, 2009, pp. 219–232.

[42] R. Nikolaev *et al.*, "VirtuOS: an operating system with kernel virtualization," in *ACM SOSP'13*, 2013, pp. 116–132.

[43] H. Lefeuvre *et al.*, "Flexos: Making os isolation flexible," in *HotOS'21*, 2021.

[44] V. Costan *et al.*, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[45] S. Pinto *et al.*, "Demystifying arm trustzone: A comprehensive survey," *ACM CSUR'19*, vol. 51, pp. 1–36, 2019.

[46] S. Park *et al.*, "libmpk: Software abstraction for intel memory protection keys (intel mpk)," in *USENIX ATC'19*, 2019, pp. 241–254.

[47] A. Vahldiek-Oberwagner *et al.*, "Erim: Secure, efficient in-process isolation with protection keys (mpk)," in *USENIX Security'19*, 2019, pp. 1221–1238.

[48] M. Hedayati *et al.*, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *USENIX ATC'19*, 2019, pp. 489–504.

[49] M. Sung *et al.*, "Intra-unikernel isolation with intel memory protection keys," in *ACM VEE '20*, 2020, pp. 143–156.

[50] R. Wahbe *et al.*, "Efficient software-based fault isolation," in *ACM SOSP'93*, 1993, pp. 203–216.

[51] M. Y. R. Gadelha *et al.*, "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.

[52] ——, "ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution)," in *TACAS'19*, vol. 11429, 2019, pp. 209–213.

[53] A. R. Bradley *et al.*, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., 2007.