

ChatGPT, Make a Secure Malloc for me

Jeremy Singer
University of Glasgow
Glasgow, UK
jeremy.singer@glasgow.ac.uk

Zheng Wang
University of Leeds
Leeds, UK
Z.Wang5@leeds.ac.uk

Abstract

Two long-term trends have now reached maturity in the computing field: namely generative AI and memory-secure processors. In our research, we aim to exploit the synergy between these trends to use customized large language models to generate system-level memory management routines that take advantage of a range of processor features supporting memory security. In this position paper, we sketch out the research agenda for this project.

1 Motivation

Memory bugs often lead to serious security issues. According to recent studies from Microsoft [3] and Google [1], memory bugs account for 70% of critical software vulnerabilities. Such problems are frequently caused by (mis)management of dynamically allocated memory. As well as being a focus for *security*, memory allocation is also significant for *performance*. A large proportion of program execution time is devoted to low-level memory management routines.

Further, the huge power demands of dynamic RAM in cloud data centres motivate the requirement for more sophisticated, adaptive memory management techniques to improve the sustainability of virtualised workloads.

There is rapid, ongoing innovation in the hardware micro-architectural domain, adding secure extensions to mainstream processor families, including Arm and Intel. This involves new basic instructions for secure enclaves, bounded capabilities and extensions to page table metadata and protection mechanisms; preventing whole classes of exploits that have plagued systems software for decades. These new processor facilities provide fundamental mechanisms for more secure memory allocators, but their full benefits are yet to be seen due to the massive developer effort required to write or optimise a memory management codebase. For example, the recent *snmalloc* secure allocator [2] comprises

25K SLoC painstakingly developed by leading industrial practitioners over four years. Further, different vendors provide highly diverse feature sets; there is no common baseline for memory-secure operation.

Our ambitious project will develop an open-source framework to enable ‘grow your own’ memory managers to meet a set of formal user requirements encompassing security and performance. While hand-crafted high-performance libraries are readily available and widely deployed, they take immense developer effort to create—and further effort to re-target to new architectures. Our proposal takes a different approach, aiming to substitute most of the human engineer time with automated machine time.

2 Typical Processor Memory Security Features

Major processor vendors have a wide, diverse range of memory security features. This feature set is growing with each new processor family—we seem to be at an evolutionary stage where no-one is clear about the ‘fittest’ extensions to support.

Key high-level memory security features include tagging, isolation and encryption. Tagging enables probabilistic pointer authentication, reducing the likelihood of forged pointers accessing data. Arm MTE is one such technology. Isolation involves secure enclaves, where memory is inaccessible to code operating outside the enclave. Intel SGX and Arm Trustzone instantiate this concept. Encryption uses secret keys to prevent attackers from reading secret data, even if they have physical access to the encrypted memory. AMD SEV is one such technology.

While these memory-secure features are being supported by commercial processor designs, they are also the topic of active research interest. The ChERI community [5], using the Arm Morello prototype, support precise bounds checking in hardware, unforgeable pointers, sealed memory, etc. This is a more heavyweight approach than existing commercial techniques, but might influence future industry direction. Further, a number of experimental RISC-V security extensions appear to be relevant to memory security.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference’17, July 2017, Washington, DC, USA

© 2023 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

3 Synthesizing Code with Large Language Models

The eye-catching publicity around ChatGPT points to the likely successful adoption of large language models for AI-generated text. A simple request like ‘ChatGPT, make a malloc for me’ results in a simple C bump-pointer allocator implementation, which is correct for trivial, single-threaded client code. More complex requests for ‘secure memory allocators’ return code that performs software bounds checking via allocator metadata. However there is no notion of underlying processor memory-secure features in these C language allocator implementations generated by ChatGPT.

Our intention is to automatically synthesise a high-performance memory allocator using assembly instructions, OS library calls, and low-level primitives as building blocks.

Initially, we will develop a catalogue of security properties for memory allocators. While some properties will be composable, others may have complex inter-dependencies. We will develop an ontology with explicit relationships between properties. In essence, we intend to specify a set of allocator properties, relating to correctness and security.

The next stage is to map these allocator properties onto low-level processor-specific primitives like CHERI pointer permissions. We will begin with a hand-coded example mapping for the Arm Morello platform, since we have prior experience on this. Following this, we will create and/or collect secure allocator code snippets that implement appropriate features, targeting security-enhanced backends like Arm MTE and Intel SGX. This corpus of allocator code fragments will be used to train a code generator.

We will create a well-defined and reusable set of characteristics for runtime memory allocation code, forming the basis for both a taxonomic classification of allocators and a training set for a DRL framework. We will formulate the problem as a single-player game using deep reinforcement learning (DRL). Given a memory allocator API that defines the inputs and outputs, the player (computer) learns to select and combine low-level primitives including, e.g. capability-based bounded buffers, coloured pointers), assembly instructions, and OS library calls, to produce a correct and effective memory allocator library.

We will build and train a DRL framework from open-source memory allocators and our training corpus. The trained DRL can then generate new memory management libraries for new security properties or hardware. Our DRL consists of a learning algorithm and a representation function. The learning algorithm will use DRL and stochastic search optimisation algorithms like Monte Carlo Tree Search to navigate the program synthesis space. The representation function then maps a (partially) generated program into a status representation to guide the search. The search can start from either an empty algorithm or a standard memory allocator implementation. It then iteratively amends the (partially)

generated code by changing, adding or removing an instruction, primitive, or library call. At each step, the current status representation is used to predict the next action as being one of the valid code transformation options or a stop action to complete the optimisation. A reward function then estimates the performance gain so far to provide feedback to guide the code synthesis process.

4 What about Correctness?

Our radical alternative approach involves specifying the memory management attributes required (including security properties) and then synthesising performant memory management code to satisfy these specifications for a particular target processor architecture. We will ensure such code is *correct by construction* by using model checking techniques to verify the generated code respects conventional malloc invariants and enhanced security requirements. Model checking has been applied successfully to verify properties of garbage collection algorithms [6]. In a similar way, we will translate our high-level specifications to formal properties, expressed in linear-time temporal logic (LTL).

Then we need to create an automated static analysis pass to generate abstract models from allocator source code written in C and the corresponding assembly code. The abstract model only captures relevant semantics from the source code, relating to dynamic memory access. We will extend techniques for abstracting models from C code [4].

Based on our previous experience with formalizing systems software, we expect the development will be exploratory and, at least initially, highly iterative. Once we have reasonably representative properties and abstract models, we can commence model checking in earnest. In itself, this is a valuable research contribution and should yield useful publications. However our main goal is to link the model checking with the code synthesis component and to support reasoning at the assembly instruction level using high-level abstractions.

Effectively, we will use formal verification to provide increased confidence that generated code is correct and matches the security specification. Verification can be used in a *feedback loop* for the code synthesis process to revert a partially generated program to an error-free status so that code search does not need to start from scratch. Since our primitives are correct-by-construction and their semantics are known to the verification system, we can focus on verifying the interfaces of the primitives and their interaction. Doing so can significantly reduce the problem space. We will also automate this verification process and use input-output behavioural synthesis to quickly reject illegal code synthesis options to accelerate verification.

References

- [1] Chromium Project. Memory safety, 2020. <https://www.chromium.org/Home/chromium-security/memory-safety/>.

- [2] Paul Liétar et al. Snmalloc: A message passing allocator. In *ISMM*, 2019.
- [3] Microsoft Security Response Center. A proactive approach to more secure code, 2019. <https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/>.
- [4] Martin Sulzmann and Axel Zechner. Model checking dsl-generated c source code. In *International SPIN Workshop on Model Checking of Software*, 2012.
- [5] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An introduction to cheri. Technical Report UCAM-CL-TR-941, University of Cambridge, September 2019.
- [6] Bochen Xu et al. A model checking framework for a new collector framework. In *Proc. MPLR*, 2022. to appear.