Memory Safety Preservation for WebAssembly

Marco Vassena
CISPA Helmholz Center for Information Security

Marco Patrignani
Stanford University
CISPA Helmholz Center for Information Security

Abstract

WebAssembly (Wasm) is a next-generation portable compilation target for deploying applications written in high-level languages on the web. In order to protect their memory from untrusted code, web browser engines confine the execution of compiled Wasm programs in a *memory-safe* sandbox. Unfortunately, classic memory-safety vulnerabilities (e.g., buffer overflows and use-after-free) can still corrupt the memory *within* the sandbox and allow Wasm code to mount severe attacks. To prevent these attacks, we study a class of secure compilers that eliminate (different kinds of) of memory safety violations. Following a rigorous approach, we discuss memory safety in terms of hypersafety properties, which let us identify suitable secure compilation critera for memory-safety-preserving compilers. We conjecture that, barring some restrictions at module boundaries, the existing security mechanisms of Wasm may suffice to enforce memory-safety preservation, in the short term. In the long term, we observe that certain features proposed in the design of a memory-safe variant of Wasm could allow compilers to lift these restrictions and enforce relaxed forms of memory safety.

1 Introduction

WebAssembly (Wasm) has gained traction as the new portable compilation target language for deploying on the web applications written in high-level languages like C, C++, and Rust. Fruit of an unprecedented collaboration between four major browser vendors, Wasm ensures that even buggy or malicious code downloaded from untrusted sources can be executed safely in a web browser [15]. To enforce security, Wasm programs are validated (type-checked) first and then executed inside a sandbox that isolates untrusted code from the browser. *Memory safety* is key to the isolation mechanism of the sandboxed execution environment: well-typed programs cannot corrupt the memory outside the sandbox (e.g., the Javascript virtual machine). Unfortunately, Wasm is still far from secure: buffer overflows and use-after-free can still corrupt the memory of a program *within* the sandbox, opening the door to attacks like cross-site scripting and remote code execution [21]. The presence of memory vulnerabilities in Wasm thwarts the strenuous efforts devoted into securing unsafe languages like C [5, 20, 22, 23, 27] and developing resource-aware memory-safe languages like Rust [17, 18]. Current compilers (e.g., Emscripten) do not attempt to protect compiled programs from Wasm-level attackers exploiting well-known memory vulnerabilities. Following the principled tradition of *secure compilation* [1, 4, 26], we propose to strengthen the Wasm compilation chain with a provably secure memory-safety-preserving compiler.

Fortunately, several aspects of Wasm promote rigorous reasoning and help us in our study. In particular, Wasm (1) has a (mostly) deterministic formal semantics that rules out undefined behaviour and (2) is type-safe [15]. The specification of Wasm has even been mechanized and verified [29]. Furthermore, the existing security mechanisms of Wasm reduce the attack surface available to target level attackers and thus simplify the job of our secure compiler. Wasm features *structured control-flow* and separates code and data memory segments, which, in combination, enforce coarse-grained control-flow integrity [2, 3] removing by construction classic stack-smashing and return-oriented programming attacks. In addition, Wasm provides state and memory *encapsulation* through modules, which represent natural boundaries where to enforce security [15].

Assuming some degree of freedom when setting module boundaries, we believe that a secure compiler could reuse the existing mechanisms of Wasm to enforce memory safety at the target level, in the short term. However, this approach rests on a strong assumption, namely that the compiler has direct control over how code gets compartmentalized. As this may not always be the case, and thus for a long-term solution, we draw inspiration from Memory Safe WebAssembly (MS-Wasm), a recent design proposal for extending Wasm with hardware-supported progressive memory-safety capabilities [11]. A secure compiler relying on MS-Wasm language-level support for memory-safety enforcement could allow looser module boundaries.

In the rest of this short paper, we discuss what notions of memory safety we wish to enforce and how to formally express them as (hyper)properties.[1] Then, we outline MS-Wasm

[1]Properties are defined over single runs of a program, while hyperproperties involve multiple runs [7].

Conference'17, Washington, DC, USA
© 2016 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

and argue that it is a suitable target candidate for secure compilation. Finally, we discuss which secure compilation criterion to use when preserving memory safety to MS-Wasm.

2 Memory Safety as a (Hyper)Property

Establishing rigorous security guarantees for our compiler requires a formal definition of memory safety, an intuitive notion that has been surprisingly hard to pin down [16]. The exact definition of memory safety has important ramifications for our work because it determines what class of security properties our compiler has to preserve and thus what protection mechanisms are needed [4].

Previous works on safe variants of C [5, 20, 22, 23, 27] treat memory safety as a simple safety property enforceable by *reference monitors* [28] that detect specific memory violations (e.g., accessing freed memory or an array out-of-bounds). Seeking a definition that trascends bad behaviours, Azevedo de Amorim et al. [6] associate memory safety with reasoning principles about state akin to noninteferference [14]. Since non-interference relates *pairs* of executions, their definition ascribes memory safety to the class of 2-hypersafety [7], which is arguably harder to preserve *robustly* than safety [4, 25].

Here, we consider a notion of memory safety based on *color tags*, inspired by a line of work on micro-policies for tag-based security monitors [8, 10]. Briefly, memory locations and pointers are tagged with colors and a memory violation occurs when a pointer accesses memory tagged with a different color. Unlike the definitions of the works mentioned above, this safety property is trace-based and agnostic to the specific semantics of the languages involved and their syntax—the trace only contains memory relevant actions (i.e., memory allocation, free, read and write). Furthermore, this definition let us study various relaxation of memory safety that could describe precisely the progressive guarantees of MS-Wasm, including spatial, (relaxed) temporal safety[2] and pointer integrity, as well as novel properties that considers only data integrity.[3]

3 Memory Safe WebAssembly

Memory Safe WebAssembly (MS-Wasm) is an extension of Wasm designed to capture sufficient metadata about pointers and memory regions to enforce memory safety efficiently, levereging dedicated hardware. In particular, MS-Wasm promotes a *progressive enforcement* of memory safety, i.e., depending on application-specific security-performance trade-offs and what particular hardware is available, the same abstractions can enforce "weaker" forms of memory safety.

[2]Relaxed temporal safety allows memory accesses through dangling pointers as long as the memory pointed to has not been reallocated [11].
[3]To reduce the overhead of enforcing memory-safety, some tools support modes that check only memory writes [12, 13, 19].

The core features of MS-Wasm design are *segment memories*, i.e., linearly addressable, zero-initialized, manually managed extents of memory, and *handles*, i.e., possibly-corrupted (forged) pointers enriched with bounds metadata. To enforce memory safety, MS-Wasm restricts the interaction between segments and memories appropriately (e.g., only handlers can access segments, provided that they point within their bounds).

In order to use MS-Wasm as a target language in our secure compilation chain, we have to first formalize its design and semantics. Then, using variations of our trace-based definition of memory safety from above, we intend to prove its progressive memory-safety guarantees involving spatial, relaxed temporal safety, and pointer integrity, and establish their relative strengths. With the help of MS-Wasm abstractions, we are then going to design a class of secure compilers that preserve clearly-defined notions of memory safety.

4 Secure Compilation to MS-Wasm

To establish the security guarantees of our compilers, we prove that they attain a secure compilation criterion. Then, to further clarify their security guarantees, we consider general compilation criteria that preserve whole classes of security properties (including memory-safety), instead of using an ad-hoc criterion. Given that we can express memory safety as a safety property as well as a 2-hypersafety property, we adopt two of the robust compilation criteria proposed by Abate et al. [4], namely *Robust Safety Property Preservation* (RSP) and *Robust 2-Hypersafety Preservation* (R2HSP). Intuitively, these criteria require compilers to preserve (hyper)properties of source programs even when they are compiled and linked with arbitrary target code, thus protecting robustly against all *active* target-level attackers. In practice, equivalent *property-free* characterizations simplify significantly the proofs of robust criteria preservation [4]. Specifically, for any compiled program and target context triggering a bad behaviour, we have to find a corresponding source-level context that produces the same bad behaviour. To reconstruct suitable source-level contexts systematically, we can apply known proof techniques based on *backtranslation* [4, 9, 24, 25].

The proofs of RSP and R2HSP differ mainly over the *kind* of bad behaviours involved, which are determined by the properties that they preserve (safety and 2-hypersafety). Since safety is a simple property, bad behaviours are just finite traces (prefixes) in RSP. In R2HSP, bad behaviours consist of pair of prefixes because 2-hypersafety is just a generalization of safety to a 2-hyperproperty [7]. By including all memory-relevant actions in our traces, we gain confidence that the criteria above characterizes correctly the class of memory-safety-preserving compilers that we intend to study.

Acknowledgements: This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

[1] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, July 2012. ISSN 1094-9224. doi: 10.1145/2240276.2240279. URL http://doi.acm.org/10.1145/2240276.2240279.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ICFEM'05, pages 111–124, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29797-9, 978-3-540-29797-0. doi: 10.1007/11576280_9. URL http://dx.doi.org/10.1007/11576280_9.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009. ISSN 1094-9224. doi: 10.1145/1609956.1609960. URL http://doi.acm.org/10.1145/1609956.1609960.

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32th Computer Security Foundations Symposium*, CSF 2019, June 2019.

[5] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of c code executing in an unverified context. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 581–594, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676972. URL http://doi.acm.org/10.1145/2676726.2676972.

[6] Arthur Azevedo de Amorim, Cătălin HriŢcu, and Benjamin C. Pierce. The meaning of memory safety. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 79–105, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89722-6.

[7] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393. URL https://www.cs.cornell.edu/~clarkson/papers/clarkson_hyperproperties_journal.pdf.

[8] A. A. d. Amorim, M. DÃĪnÃĺs, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830, May 2015. doi: 10.1109/SP.2015.55.

[9] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[10] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694383. URL http://doi.acm.org/10.1145/2694344.2694383.

[11] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, pages 4:1–4:8, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7226-8. doi: 10.1145/3337167.3337171. URL http://doi.acm.org/10.1145/3337167.3337171.

[12] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4241-4. doi: 10.1145/2892208.2892212. URL http://doi.acm.org/10.1145/2892208.2892212.

[13] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/stack-object-protection-low-fat-pointers/.

[14] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982. doi: 10.1109/SP.1982.10014.

[15] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062363. URL http://doi.acm.org/10.1145/3062341.3062363.

[16] Michael Hicks. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/, 2014. Accessed: 2019-10-16.

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017. ISSN 2475-1421. doi: 10.1145/3158154. URL http://doi.acm.org/10.1145/3158154.

[18] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3217-0. doi: 10.1145/2663171.2663188. URL http://doi.acm.org/10.1145/2663171.2663188.

[19] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542504. URL http://doi.acm.org/10.1145/1543135.1542504.

[20] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40, June 2010. ISSN 0362-1340. doi: 10.1145/1837855.1806657. URL http://doi.acm.org/10.1145/1837855.1806657.

[21] NCC Group Whitepaper. https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf, 2018. Accessed: 2019-10-11.

[22] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005. ISSN 0164-0925. doi: 10.1145/1065887.1065892. URL http://doi.acm.org/10.1145/1065887.1065892.

[23] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005. ISSN 0164-0925. doi: 10.1145/1065887.1065892. URL http://doi.acm.org/10.1145/1065887.1065892.

[24] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *21st ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 103–116, 2016. doi: 10.1145/2951913.2951941. URL https://www.williamjbowman.com/resources/fabcc-paper.pdf.

[25] Marco Patrignani and Deepak Garg. Robustly Safe Compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019*, ESOP'19, 2019.

[26] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation a survey of fully abstract compilation and related

work. *ACM Comput. Surv.*, 51(6):125:1–125:36, January 2019. doi: 10.1145/3280984. URL https://doi.org/10.1145/3280984.

[27] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with checked c. In *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17138-4.

[28] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000. ISSN 1094-9224. doi: 10.1145/353323.353382. URL http://doi.acm.org/10.1145/353323.353382.

[29] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5586-5. doi: 10.1145/3167082. URL http://doi.acm.org/10.1145/3167082.