Security Properties for Stack Safety

ANONYMOUS AUTHOR(S)

The call stack is a perennial target for low-level attacks, leading to consequences from leakage or corruption of private stack data to control-flow hijacking. A profusion of software and hardware protections have been proposed for detecting or preventing such attacks, including stack canaries [2], bounds checking [5, 9, 10], split stacks [7], shadow stacks [3, 12], capabilities [1, 6, 13–15], and hardware tagging [11]. The protections offered by such mechanisms are commonly described in terms of concrete examples of attacks that they can prevent. At best, they define stack safety by reference to an idealized machine that is arguably stack safe by construction [14]. But these mechanisms can be intricate, and it would be useful to have a precise, generic, and formal specification for stack safety, both to compare the security claims of different enforcement techniques and to rigorously validate such claims.

We propose such a characterization using the tools of language-based security: The informal claim that "stack safety protects a caller from its callee" amounts to saying that it guarantees the *integrity* and *confidentiality* of the caller's local state until it regains control.

We formalize these integrity and confidentiality requirements as trace (hyper-)properties, with two different variants: *stepwise* variants, in which a caller's data must *never* be read or modified during a call, and *observational* ones, in which callees may read from and write to their caller's stack frame, as long as these "risky" behaviors do not affect the system's observable behavior. The observational properties are more extensional, and any reasonable protection mechanism ought to enforce them, even if it does not prevent every single dangerous read or write.

Confidentiality is especially interesting, as it is based on the traditional notion of noninterference. But where noninterference is normally presented as an end-to-end hyper-property, stack confidentiality is noninterference applied over multiple nested subtraces – each individual call's behavior must be invariant regardless of the state of the stack at its entry.

This formulation not only captures the intuition that the callee cannot directly access the caller's state, but gives a novel way of looking at control-flow attacks. We are not interested in attacks that merely aim to execute arbitrary code – in fact, our attacker model assumes that the (attacking) callee may already execute arbitrary code. By extension, we do not place any particular restrictions on control flow. Perhaps suprisingly, we can still meaningfully distinguish the caller from its callee, and talk about protecting the caller's data. After a call, control is considered to belong to the callee until the trace reaches a valid "return target," typically a state with the program counter at the next instruction and the stack pointer restored. If the caller's data are accessed before then, it is a violation of integrity or confidentiality.

These properties can optionally be extended with a notion of *well-bracketed control flow* as in Skorstengaard et al. [14], the system-wide control-flow property that callees always return to their immediate caller, if they return at all. This property is largely orthogonal to our data-protection properties.

To demonstrate the utility of our properties, we use them to evaluate an existing mechanism, the *stack-safety micro-policies* of Roessler and DeHon [11], re-implemented in the Coq proof assistant on top of a RISC-V specification. We use QuickChick [4, 8], a property-based testing tool for Coq, to generate random programs and check that Roessler and Dehon's micro-policies correctly abort programs that would violate stack safety. Furthermore, we check incorrect enforcement variants (both variants that we accidentally created during our re-implementation of the micro-policy and ones that we intentionally crafted to be broken in order to increase our confidence in testing and the enforcement mechanism itself). The testing framework is able to generate counterexamples that violate our properties but are not halted by incorrect enforcement mechanisms, increasing our confidence when it finds no counterexamples under the real enforcement mechanisms.

Our testing supports the stack-safety claim of Roessler and Dehon's *Depth Isolation* micro-policy, in which memory cells within each stack frame are tagged with the depth of the function activation that owns the frame and access to those locations is then permitted only when that activation is currently executing. On the other hand, we find that their *Lazy Tagging and Clearing* policy violates the temporal aspect of confidentiality in corner cases where data can leak across repeated calls to the same callee, and also violates integrity if the leak happens to use the caller's frame. We propose a variant of *Lazy Tagging and Clearing* that should enforce observational integrity and confidentiality, albeit at some performance cost. The next step is to expand testing to these properties and validate the variant; efficiently testing observational properties presents unique challenges because a property violation can be arbitrarily far removed from the initial "risky" read or write.

Finally, we demonstrate our model's flexibility by extending it to different settings. First, we allow variables to be passed on the stack, allowing some of the state of the caller to be shared with the callee (and potentially with a further callee, and so on.) Then we expand the number of stacks as we move to a simple coroutine model, albeit one with statically bounded regions for each stack.

In addition to testing lazy tag-based policies against our observational properties, we are adapting the uninitialized capability model of Georges et al. [6] to work with our testing framework for validation against our properties. Of the several Cheri-based stack protection schemes, its treatment of unitialized frames seems most compatible with our model of confidentiality, without the need for expensive stack clearing between calls. We expect that Cheri-based techniques support our properties if we assume extra cooperation from callers, namely that they do not leak their capabilities to the attacker. There is no equivalent dynamic requirement in a tag-based system because the policy can dynamically enforce cooperation.

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

REFERENCES

[1] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (San Antonio, Texas) *(SSYM'98)*. USENIX Association, USA, 5.

[3] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. Association for Computing Machinery, New York, NY, USA, 555–566. https://doi.org/10.1145/2714576.2714635

[4] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2014. QuickChick: Property-Based Testing for Coq (abstract). In *VSL*. http://www.easychair.org/smart-program/VSL2014/index.html

[5] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–114. http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf

[6] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

[7] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 147–163.

[8] Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*. Electronic textbook. Version 1.0. http://www.cis.upenn.edu/ bcpierce/sf.

[9] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 245–258. http://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports

[10] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *9th International Symposium on Memory Management*. ACM, 31–40. http://acg.cis.upenn.edu/papers/ismm10_cets.pdf

[11] Nick Roessler and André DeHon. 2018. Protecting the Stack with Metadata Policies and Tagged Hardware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 478–495. https://doi.org/10.1109/SP.2018.00066

[12] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175

[13] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 5 (Dec. 2019), 53 pages. https://doi.org/10.1145/3363519

[14] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

[15] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 457–468.