A Security Analysis of Labeling-Based Control-Flow Integrity Schemes

David Demicco, Matthew Cole, Shengdun Wang and Aravind Prakash
Binghamton University
email:{ddemicc1,mcole8,swang206,aprakash}@binghamton.edu

***Abstract*—Secure and transparent policy enforcement by a cloud provider is crucial in cloud infrastructures. Particularly, enforcement of control-flow integrity (CFI) policy has been widely accepted for stopping software-induced attacks. Using low-level hardware metadata to encode CFI policy is a fairly recent development. Besides moving enforcement out of the software and into the hardware for performance benefit, tagging metadata also offers other benefits in the precision of defenses. We evaluate several different metadata layouts for CFI policy enforcement, and examine the layouts' effects on the number of valid forward edges remaining in a RISC-V binary after policy enforcement. Additionally we look at related work in tag-based tools that provide CFI policy enforcement in order to get a sense of their performance and the design trade-offs they make. We evaluate our policy and the related works in terms of space and precision trade-offs for forward- and backward-edge CFI, finding that some trade-offs have a higher impact on the number of remaining forward edges, notably return address protection. Additionally, we report that existing backward edge protections can be highly effective, reducing the number of remaining backward edges in a protected binary to an average of 0.034% over an equivalent coarse-grained CFI.**

***Index Terms*—tagging architectures, control flow integrity, RISC-V architecture, binary analysis**

I. Introduction

Tagging is a technique that applies metadata to data and code such that the metadata enacts a usage policy to that data, similar to the capabilities model used in operating systems. This usage policy typically has two halves: one half applied to the instructions specifying what an instruction should have the capability to do, and the other half applied to data specifying how the data should be used. Combined, the two tags create a "user-usage" relationship that can be monitored by specially modified hardware as part of the instruction pipeline. Tagging schemes are beneficial in two ways. First, they capture rich program semantics that are typically lost during compilation. Second, they can be directly consumed by the hardware and therefore provide transparency and performance benefits over software monitors [1].

In order to achieve these benefits, the tagging scheme must be thoughtfully designed. Firstly, the designer must consider the number of tags necessary to capture the desired policy because this determines the number of bits required to encode the tag. Secondly, the designer must choose the tag coverage, whether a particular tag will apply to more than one instruction or data item. Finally, they must decide whether multiple tags will be compressed to fit within the quantum of addressable space (e.g. byte addressable memory, or instruction width in fixed-width instruction sets). Combined, these decisions inform a design trade-off of greater or fewer number of bits per tag, which has implications on the system as a whole: more bits per tag comes at the cost of binary size increase, performance overhead of processing wider tags, and pollution in one or both of the data and instruction caches. Because a designer may not be able to best decide this trade-off for all users, they may wish to provide user-parameterized tagging schemes instead of a fixed parameterization.

These trade-offs become clear when examining several existing control-flow integrity (CFI) tagging schemes. *Equivalence classes* are the set of destinations reachable from a single control flow transfer, that is, these destinations would check for a single common CFI label. Generally as tag width increases, the number of possible CFI labels increases, and thus the number of classes that can be labeled. If the tagging scheme does not allocate sufficient width per tag, it might not be able to handle the number of equivalence classes needed by a particular program, with the result that one or more equivalence classes must be congealed into a single super-class, causing a concomitant decrease in security provided. Conversely, if the tagging scheme allocates too wide of a tag, it does not provide any additional security benefits, and does so with the design cost of wider tags such as binary size and/or cache pressure increases. We measured the number of forward and backwards edges that form an equivalence class for a particular configuration of several CFI tagging schemes. Through these measurements, we wanted to find an optimization in the design space for both the number of equivalence classes and the density of functions across these equivalence classes. Intuitively, the greater the density of functions in a particular equivalence class, the greater the degree of freedom that an attacker has to construct a code-reuse attack that evades the CFI defense.

Through our work, we explore these design trade-offs and make the following contributions:

- We evaluate various tag widths for encoding CFI labels for a variety of programs. We find that there is a tendency for a few equivalence classes (i.e. function signatures) to dominate the distribution of labeled functions.
- We evaluate PUMP [2], ZERØ [3], and RetTag [4], providing a quantitative metric for measuring additional CFI precision per additional tag bit.
- We argue that there is a point of diminishing returns,

where increasing the number of tag bits does not significantly increase fidelity of a tag-based CFI labeling scheme.

Relevance in Cloud Scenarios: Modern cloud infrastructures (e.g. Azure, AWS) heavily depend on customized systems and hardware that can effectively enforce security in a manner transparent to the hosted applications. A thorough and in-depth evaluation of available protection mechanisms (such as CFI) is a pressing need. Our work sheds light on the cohesive interplay between software (tag policy and CFI labeling) and hardware (tag and CFI enforcement) layers and the resulting security benefits that are directly applicable in cloud scenarios. Current and future cloud infrastructures (such as those built on RISC-V) can readily benefit from the outcomes of our work.

II. Background

A. Control-Flow Integrity

Control-flow integrity is a security property that requires a program's execution to follow a Control-Flow Graph (CFG) specified ahead of time. The CFG is comprised of control-flow transfer edges between two instruction vertices, and can be enforced at the site of each *forward-edge* indirect control-flow transfer and *backwards-edge* return. CFI's enforcement usually occurs by in-lined instrumentation. At each call site, a label specifies which equivalence class of destinations is permissible. Then at the call's destination, a label ID is encountered specifying to which equivalence class that block of code belongs. If they match execution continues, but if they do not match then the monitor will abort execution because it has detected a malicious control flow.

A variety of metrics exist to quantify the efficacy of a CFI defense, for example average gadget length [5], [6], gadget reduction [7], backwards-edge return targets, [8], and Average Indirect Branch Reduction (AIR) [7], [9], [10]. In this paper, we focus on the number of remaining forward/backward edges after the application of a defense.

B. Tagging Schemes

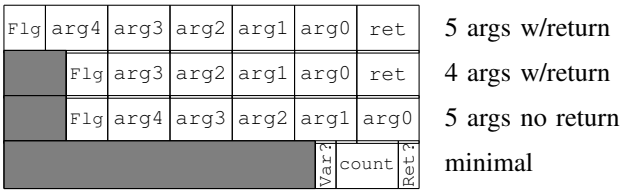
Tagging is the process of associating a piece of metadata – the tag – with a piece of information (data or code). This tag defines how the information should be used, whether applied to an instruction or a data object. At run-time, special tag-aware hardware ensures that the instruction's execution satisfies its tag, and a data object's run-time type is consistent with its type.

Tagging with CFI support incorporates CFI *labels* that define allowable source-destination pairs for indirect control-flow transfers. *A unique CFI label indicates which specific indirect call instruction(s) can call which specific function(s).* Practical CFI solutions depend on hardware protections to ensure that code is not writable. This allows CFI to focus on indirect and not direct control flow transfers. The key difference between a CFI label and a tag is that the label is an identification for CFI edges, whereas a tag can capture other metadata (e.g. type information). Some defenses implement CFI labels using hardware tagging, and in these cases labels and tags may appear to be synonymous (i.e. PUMP [2] and ZERØ [3]). The number of bits available to express tags directly correlates to the quality of CFI defense. Too few bits mean collisions in CFI labels, and too many imply unused bits that increase bloat.

III. Evaluation Methodology

We evaluate multiple implementations of CFI schemes and their impact on the overall security of the system. We use labeling for forward-edge protection and light-weight data tagging for return address protection.

(a) `lui` instruction



(b) Example `immediate` field values

Fig. 1: Bitfield diagrams for the label parameterizations. Width of argument count and argument/return type fields are three bits, `Flg` is two bits, and `Var?`/`Ret?` flags are 1 bit each. Gray regions are bits obligatory to the `immediate` field but unused by this label scheme.

We evaluate labels based on one of two parameters: the type of the argument, and the size of the argument. For *type-based labels*, we consider the type of the argument as one of 8 types: *void, 8, 32, and 64 bit integers, floats, arrays, pointers, and other*. Note that because we have eight types, we need three bits per argument to encode its type. For *size-based labels*, we use the argument's `sizeof(argument)`, but with two special cases: `void` arguments are 0 (i.e. there is no return type or no argument), and values whose size cannot be statically determined are value 7 ("unsized"). This gives 8, 16, 32, 64, 128, and 256 as the possible argument sizes.

A. Parameterized Tagging Schemes

We consider the following specific parameterizations of labeling schemes, shown in Figure 1:

- Type-based labels, with a generic pointer type. 5 arguments and return.
- Size-based labels. 5 arguments and return.
- Size-based labels 4 arguments and return.
- Size-based labels, 5 arguments and no return.
- Type-based labels, with a generic pointer, and size-based integers. (e.g. `i8`, `i32`, `i64`).
- Minimal label. Here, we count number of arguments (up to seven), and append a `1` if there is a non-void return

type, or a `0` if there is a void return type. We use a total of five bits: one bit for the variadic function signature flag, three bits for the arguments count, and one bit for the return type flag.

Additionally, for all parameterizations we add a 1-bit flag for variadic function signatures (i.e. the final argument is a `varargs`-receiving argument).

These schemes were carefully selected to investigate different design trade-offs a scheme designer would have to make. For type based labels, we needed a generic pointer to handle cases where pointers are cast between different types. Size-based labels are agnostic to the type, but still allow differentiation between sizes of the underlying types. This addresses a data attack that might seek to over-read or overwrite the bounds of the object pointed to by the argument. We consider 8 possible types, and 8 possible sizes requiring 3 bits to encode. Therefore, we need 15 or 12 bits of space in the arguments, plus another three bits of space for the return type. We select the RISC-V `lui` instruction's `immediate` field to store our label. We consider five, then four arguments to show the effect upon precision of decreasing the number of arguments contributing to the label encoding. We also consider a label that does not include the return type to examine the effects on precision by incorporating return type information in label encoding. We consider a fifth scheme with a coalesced pointer type but discriminated integer types because we observe that functions taking pointer arguments usually do so with `void*` type specified. We consider the minimal scheme to investigate how much coverage and performance we can achieve while dedicating as little space as possible for label encoding.

B. CFI Tagging

a) Forward-Edge Labeling: We add a function labeling pass in the optimizer, changing the pass output for each scheme to handle the differing number of arguments and flags that may be present in the label. Ultimately, the function labeling pass produces a single integer value representing the label for each computed jump, address taken function, or externally available function. This value is then emitted at the appropriate location when a machine function is lowered into the binary as part of an idempotent `lui` instruction.

b) Label Encoding: as discussed in Section III-A we selected the idempotent `lui x0` instruction due to its large immediate field. This provides backwards compatibility and 20 bits of label space.

c) Backward-Edge Tagging: For backwards-edge tagging, we add metadata tags to the return address push and pop instructions with a return address usage (`RA`) tag, marking the return address in memory, and allowing only instructions with the return address usage tag to access the return address in memory.

IV. Experimental Results

A. Dataset

Along with the labeling schemes presented in Section III, we consider three adjacent tools: PUMP [2], ZERØ [3], and RETTAG [4]. We summarize both our label schemes and the schemes of these tools in Table I. PUMP has a forward-edge CFI that operates on a per-function labeling scheme requiring a complete program CFG that has been verified [11]. For backward-edge, PUMP protects the return address on the stack[12]. To do this it uses either a lightweight return address protection scheme that marks return address in memory with a single bit tag (we consider this as light hardware backing), or a more powerful "static authorities" model, which protects objects on the stack from access and use from outside the stack frame by tagging them with an ID for the current frame, and restricting access based on that tag. ZERØ's CFI is a subset of its pointer-flow integrity (PFI) model and encodes each unique function type signature into 10 unused bits in the pointer itself. The width of the unused bits limits ZERØ to 1024 unique function pointer types.

B. Experimental Setup

For this evaluation on our own labels, we used a modified LLVM-10 RISC-V compiler to build musl-libc for each label type variation specified in Section III. We then statically link to that specific version of labeled musl-libc while building a subset of the SPEC CPU2017 benchmark suite using the same modified compiler at O0 and O2. Once completed, we load each of these binaries into the Ghidra tool for data collection and analysis. For our analysis we only use information that the compiler has placed into the compiled binary and that Ghidra is able to access.

C. Forward Edge Data Collection

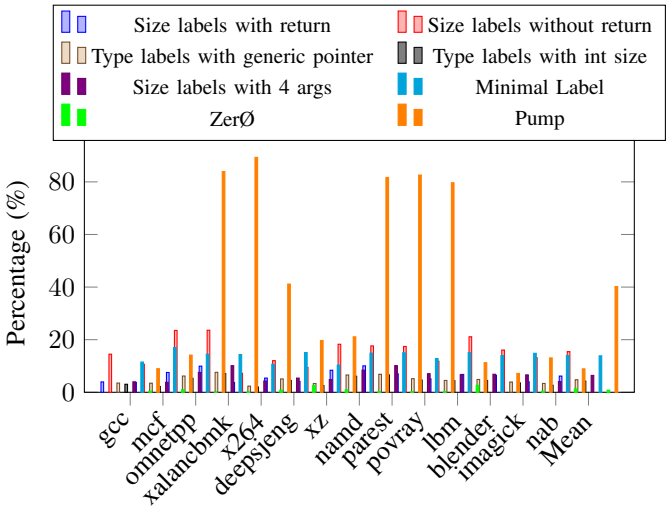


Fig. 2: Remaining forward edges for O2 optimization, defined as the fraction of valid forward edges remaining when compared to a coarse-grained CFI.

For the baseline (without CFI), we count any computed call site as being able to be re-purposed to target any instruction, and draw a possible forward edge between them

Label Scheme	Protection	Bits	Description
Type, 5 args	Forward	20	Function's first 5 arguments' and return value's type
Size, 5 args	Forward	20	Function's first 5 arguments' and return value's size
Type, integer size	Forward	20	Function's first 5 arguments' type; integer return values are differentiated by size
Size, 4 args	Forward	17	Function's first 4 arguments' and return value size
Size, no return	Forward	17	Function's first 5 arguments' size; no return value considered
Minimal	Forward	5	Variable argument flag, number of arguments, void/non-void return flag
Tool	**Protection**	**Bits**	**Description**
ZERØ	Forward	10	Stores a label for each unique function pointer, checks label on calls
	Backward	1	Uses tag in memory to prevent overwrite of return address
RETTAG	Backward	16	Stores *PAC* calculated from stack pointer and unique function ID
PUMP	Forward	*variable*	Stores source and destination ID in metadata, compares to CFG edges
	Backward (Type 1)	1	Uses tag in memory to prevent overwrite to return address
	Backward (Type 2)	*variable*	Uses *static authorities*. See Section IV-A

TABLE I: Table of schemes and tools evaluated in this work. **Protection** describes whether the tool provides forward or backward edge protection. **Bits** is the number of encoding bits used. *Variable*-sized tags in PUMP are generally at least pointer-sized per memory word, but can be compressed under certain optimized conditions.

(i.e. |forward edges| = |computed calls| × |instructions|). For the tag-only forward edge numbers, we count any computed call site as being able to target any valid function entry point. The call site must have a call tag, and the target of a call must have a target of a call tag (i.e. |forward edges| = |computed calls| × |functions|). We collect the label-based forward edge data using Ghidra to iterate over the computed call sites in a compiled binary, retrieve the labels placed in `nop` instructions preceding the call sites, and create a list of caller labels. We follow this by iterating over all functions that have received a label, and counting edges between call-sites and functions that have a matching label. We compare this number to the reduction from the tag-only CFI, quantifying improvement over a basic coarse-grained approach for each variation.

a) PUMP and ZERØ: For the forward edge reduction results of PUMP and ZERØ, we use their reported results. To estimate ZERØ's reduction, we create one function label per unique function signature to use the label emitted to the binary. We then proceed as described above, matching labels on computed call sites with labels on functions to generate the number of forward edges remaining. Although its authors do not provide a reference implementation, we estimate PUMP's reduction by the results of Ghidra's instruction flow analysis. Where Ghidra cannot determine the target of a computed call site, we conservatively treat any function within the binary that has its address taken as a possible target. This leads to an overestimation in our results for forward edges for the fine-grained CFI that PUMP describes. We explain this overestimation further in Section IV-E2.

D. Backward Edge Data Collection

For the baseline backward-edge data we consider any function return as having an edge with any location in the code (i.e. |backward edges| = |returns| × |instructions|). This is consistent with the attacker being able to overwrite the return address on the stack with a simple buffer overflow, and with no other protections being present. For the tag-only measurements, we observe that the next instruction executed after a return must be preceded by a call instruction (i.e

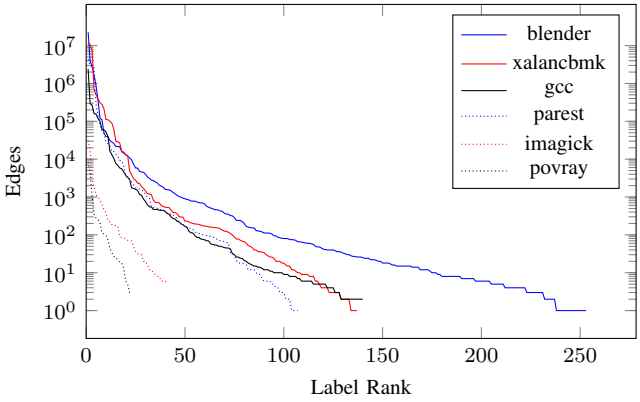


Fig. 3: Log-scaled plot showing count of forward edges for any given label corresponding to a function signature. All benchmarks were compiled with `-O0` optimization level and using the "general pointer with integer sizes" labeling scheme. Edges is the count of source-sink edges using that label. Rank is the ordering of number of edges for a function signature. In other words, a rank of 1 indicates this is the label appearing in the most edges.

|backward edges| = |returns| × |call sites|). To collect an estimate for this data in Ghidra we count the number of calls, and treat any return as having an edge with the instruction that follows any call.

a) Light Hardware Enforcement: For this evaluation, we consider hardware capable of protecting the return address on the stack from being overwritten such as ZERØ [3] and PACSTACK [13]. Because no return address can be overwritten by an attacker, they are limited to addresses already on the stack. We use the statically-determined call depth as a lower bound estimation of the number of return addresses on the stack at any given time. Gathering this count in Ghidra requires traversing the programs backward-edge call-graph through examining the code references to a function entry point. We use the depth of the call graph as an estimation for the number of valid backward edge targets from any return.

	−O0			−O2		
Benchmark	**Tag Only**	**Light Hardware**	**PUMP & RetTag**	**Tag Only**	**Light Hardware**	**PUMP & RetTag**
gcc	3.77	0.00896	0.00057	4.76	0.01363	0.00077
mcf	2.21	1.38266	0.13793	2.44	1.30556	0.13889
omnetpp	5.63	0.03510	0.00146	5.80	0.01191	0.00200
xalancbmk	5.78	0.00707	0.00078	6.12	0.00719	0.00164
x264	1.99	0.18284	0.01866	2.17	0.15689	0.02320
deepsjeng	2.15	0.76921	0.06693	2.42	0.83743	0.08036
xz	2.19	0.27583	0.04876	2.55	0.36968	0.06888
namd	3.15	0.03692	0.00187	2.56	0.31398	0.01162
parest	6.49	0.01037	0.00032	4.84	0.01709	0.00064
povray	3.80	0.06800	0.00470	3.96	0.24272	0.00556
lbm	1.93	1.62250	0.16313	2.29	1.58423	0.16207
blender	3.38	0.00712	0.00058	4.02	0.00361	0.00070
imagick	3.13	0.05338	0.00287	4.23	0.06443	0.00372
nab	2.99	0.24656	0.02873	3.56	0.22042	0.02871
Mean	**3.47**	**0.33618**	**0.03409**	**3.69**	**0.36777**	**0.03777**

TABLE II: Remaining return address edges percentages for SPEC CPU2017 C and/or C++ benchmarks compiled at two compilation levels for three levels of defense. Mean is the geometric mean of the remaining edges without weighting for the count of return addresses in each benchmark.

As with the forward edge results, we compare these numbers to the "tag only" results, not to the "no CFI" results.

b) PUMP *and* RETTAG*:* PUMP and RETTAG have powerful return address protection schemes, which also protects against overwrites or reads from return addresses in different stack frames. Due to this, the estimates for their backward edge counts are one per call-site.

E. Key Results

1) Forward Edge: We found that even basic tag protection reduces the attack surface available to an attacker by an average of 99.208% leaving just 0.792% of forward edges remaining. Using that as our baseline for all further comparison, we find the best performing label scheme out of the ones we tested was type-based labeling with integer sizes, leaving 4.772% of the forward edges remaining. This is consistent with the observation that of the varying types present in the LLVM IR, integer types are the one whose sizes differ most from function argument to function argument. The worst performing scheme depends on the program being examined, with the minimal scheme sometimes performing better than a size-based labeling scheme that does not consider the return argument. This result occurs in several of the benchmarks including omnet, blender, and povray. The full results for each scheme at `O2` optimization level are presented with comparative results for ZERØ and PUMP in Figure 2.

2) PUMP *&* ZERØ*:* By relying on Ghidra's analysis – instead of being able to examine binaries with PUMP's CFI – the need for a conservative estimation can cause the number of unknown target computed calls and the number of address taken functions to grow so large as to overwhelm the edge reduction provided by one-to-one source and target pairs. This is especially true in C++ programs that have a large number of virtual method tables and virtual function calls (a common source of address taken functions and unknown target calls, respectively). Our results for programs where these properties hold true demonstrate this flaw clearly: omnet (66.60%), povray (79.22%), and parest (62.61%). For source files with fewer address-taken functions, more accurate results can be obtained, as seen in gcc (3.34%). This is a limitation of our analysis, not PUMP's CFI scheme.

ZERØ's results are due to its unique function labels based on LLVM's function type; these do not take into account casts between pointer types. Due to this approximation, the results range from good (deepsjeng: 2.42% remaining), to very good (namd: 0.142% remaining). However this means that there will also be missed control flow edges in the program that other defensive policies must handle.

3) Label Histograms: Earlier we observed that there was little difference between the "five arguments" and "four arguments" schemes, but a large difference between the "five arguments with return" scheme, and the "five arguments without return" scheme. We plotted the count of the number of edges for each signature plotted as ticks along the x-axis in Figure 3. Here $|\text{edges}| = |\text{sources}| \times |\text{targets}|$ where sources are the call sites with a given label, and targets are the functions with that label. The logarithmic y-axis emphasizes that the number of edges is dominated by a few function signatures such as `(void)(*, *)` and `(*)(*,*)`.

4) Backward Edge Results: We found the remaining backward edges after tag-only reduction (i.e. returns must only target call preceded instructions) to be 3.55%. With light hardware backing we estimate the further reduction from the tag-only approach to have only 0.352% of backward edges remaining from the tag only approach. PUMP's and RETTAG's backward edge remainder is only 0.035%. We elaborate upon these results in Table II.

V. RELATED WORK

Memory tagging is not the only approach considered by recent papers. MORPHEUS [14] presents an cryptographic system that protects data on the stack and heap by encrypting and decrypting it during normal program operation. This approach requires churning (re-encrypting) the program at intervals

in order to keep the data safe. Protection of the backward edge has often been performed by shadow stacks [15], [16]. Shadow stacks have high costs associated with them, and much work has been done recently on alternate designs such PUMP [2], [12] and RETTAG [4], two works whose backward edge protections we do examine in this paper. Further, a recent effort codenamed STAR [1] is a full-stack defense mechanism that incorporates inlined instruction tagging as a key novelty in order to achieve superior performance. While STAR does provide forward- and backward-edge CFI defense using instruction and data tags, we do not evaluate STAR in this effort.

One comparable examination to ours is CSCAN [17], which is a generic dynamic analysis tool that attempts to measure the number of valid control flow transfers at any given indirect control transition. This dynamic approach suffers from coverage issues. Since they evaluated CSCAN on SPEC CPU2006 instead of SPEC CPU2017, we cannot directly compare it to our binary analysis using Ghidra.

Burow et al. performed a study [18] of existing CFI implementations which combined a qualitative categorization of the implementations' supported control flow transfers, static analysis precision, and performance overheads. Most similar to our own work, the authors also considered quantitative security guarantees by counting the number and measuring the size of equivalence classes after applying each of the available defenses. Our work extends their study into equivalence classes in two ways: First, we use a more modern, larger benchmark set (SPEC CPU 2017 versus SPEC CPU 2006) considering all C and C++ benchmarks in the suite instead of just four benchmarks. Second, we studied tag-based defenses not then available (PUMP, ZERØ, RETTAG) and also consider the implications of their tag specifications.

VI. CONCLUSION

We draw the following conclusions: (1) There are diminishing returns in terms of equivalency classes for using label space to track additional return arguments; (2) not taking the return address of an argument into account when constructing equivalency classes leads to large growth in remaining forward edges; (3) most of the remaining forward edges come from just a few of the equivalency classes; (4) all return edge protections are very effective at stopping attacks by reducing the number of available backward edges.

ACKNOWLEDGMENT

This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929, National Science Foundation Awards #2047205 and #2146212, and DARPA award #81192. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] R. Gollapudi, G. Yuksek, D. Demicco, M. Cole, G. Kothari, R. Kulkarni, X. Zhang, K. Ghose, A. Prakash, and Z. Umrigar, "Control flow and pointer integrity enforcement in a secure tagged architecture," to appear in 2023 IEEE Symposium on Security and Privacy.

[2] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 487–502. [Online]. Available: https://doi.org/10.1145/2694344.2694383

[3] M. T. Ibn Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "ZerØ: Zero-overhead resilient operation under pointer integrity attacks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 999–1012.

[4] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "Rettag: Hardware-assisted return address integrity on risc-v," in *Proceedings of the 15th European Workshop on Systems Security*, ser. EuroSec '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 50–56. [Online]. Available: https://doi.org/10.1145/3517208.3523758

[5] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 258–269.

[6] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *2015 Network and Distributed System Security (NDSS)*, 2015.

[7] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 577–587. [Online]. Available: https://doi.org/10.1145/2594291.2594295

[8] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[9] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[10] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[11] A. A. d. Amorim, M. Dénès, 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*, 2015, pp. 813–830.

[12] N. Roessler and A. DeHon, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[13] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 357–374. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand

[14] A. Harris, T. Verma, S. Wei, L. Biernacki, A. Kisil, M. T. Aga, V. Bertacco, B. Kasikci, M. Tiwari, and T. Austin, "Morpheus ii: A risc-v security extension for protecting vulnerable software and hardware," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2021, pp. 226–238.

[15] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," p. 555–566, 2015. [Online]. Available: https://doi.org/10.1145/2714576.2714635

[16] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," pp. 985–999, 2019.

[17] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, *Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1821–1835. [Online]. Available: https://doi.org/10.1145/3372297.3417867

[18] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, apr 2017. [Online]. Available: https://doi.org/10.1145/3054924