TypeShield: Practical Forward Edge Based Attack Protection

Paul Muntean
Department of Computer Science
Technical University of Munich, Germany
28.06.2017

Outline

- Introduction
- Background
- Overview
- Design
- Evaluation
- Conclusion

Introduction

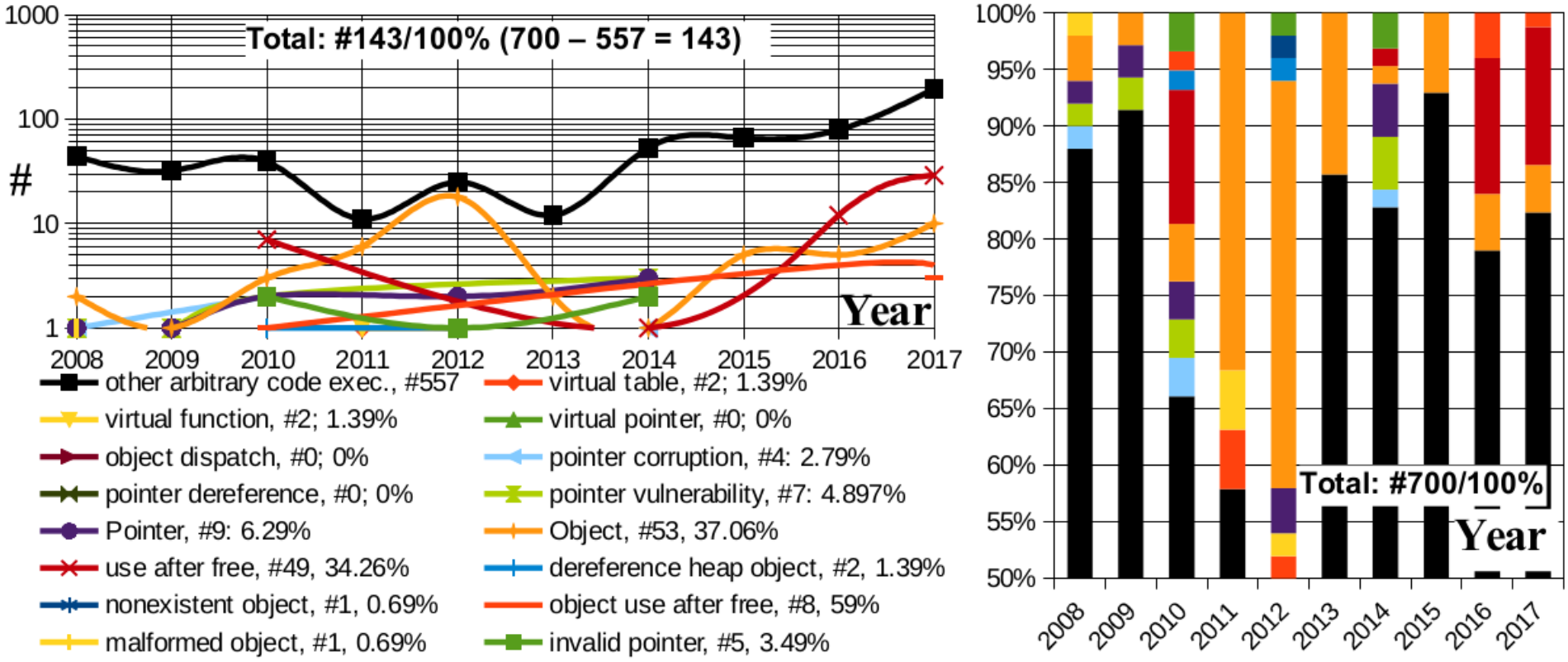

Figure 1: 10 Years of Arbitrary Code Reuse Attacks Statistics.

*just U.S. NVD data, as of May 2017

Introduction

- **What is the problem?**
  - Forward edges in binary programs can be easilly corrupted to perform code reuse attacks (CRAs)
- **What are the current solutions?**
  - A plethora of binary and source code tools for runtime forward edge protection
- **Where are the limitations of these solutions?**
  - Poor to good precision w.r.t. calltarget reduction per forward edge indirect control flow transfer
- **What is our idea/insight?**
  - Combine function parameter count and paramter types (register wideness) for a fine grained CFI-based policy for forward edge indirect control flow transfers in binaries programs
- **What are our contributions?**
  - **1.** Novel Forward Edge Based Attacks Mitigation Technique, **2.** Implemented Prototype, **3.** Security Analysis, **4.** Evaluation, **5.** Detailed Comparison, **6.** Usable Prototype and Results Reproducibility, **7.** Platform for mitigation of other CRAs

Background

- System V ABI x86-64 Bit: Parameters to functions are passed in the registers **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**, and further values are passed on the stack in reverse order
- Microsoft ABI uses 4 registers and the rest is passed over the stack
- Alias analysis in binaries is undecidable, see Ramalingam et al. TOPLAS'94 paper; one reason for CRAs
- DEP, ASLR etc. bypassed
- 10 Mb. binary → 10 Mil. poss. addresses to jump
- Call site (start of forward edge), call target (end of forward edge), not only!

```
1 class nsMultiplexInputStream final
2  :public nsIMultiplexInputStream //A0
3  ,public nsISeekableStream //A1
4  ,public nsIIPCSerializableInputStream //A2
5  ,public nsICloneableInputStream{ //A3
6 nsTArray<nsCOMPtr<nsIInputStream>> mStreams;
7 NS_IMETHODIMP nsMultiplexInputStream::Close(){
8   MutexAutoLock lock(mLock);
9   mStatus = NS_BASE_STREAM_CLOSED;
10  //set NS_OK flag
11  nsresult rv = NS_OK;
12  //get array length
13  uint32_t len = mStreams.Length();
14 //array-based main loop gadget (ML-G)
15 for (uint32_t i = 0; i<len; ++i){
16  //(1) hijacked indirect call
17  nsresult rv2=mStreams[i]->Close();
18  if (NS_FAILED(rv2)) {
19      rv = rv2;
20  }
21 }
22  return rv;
23 }
```

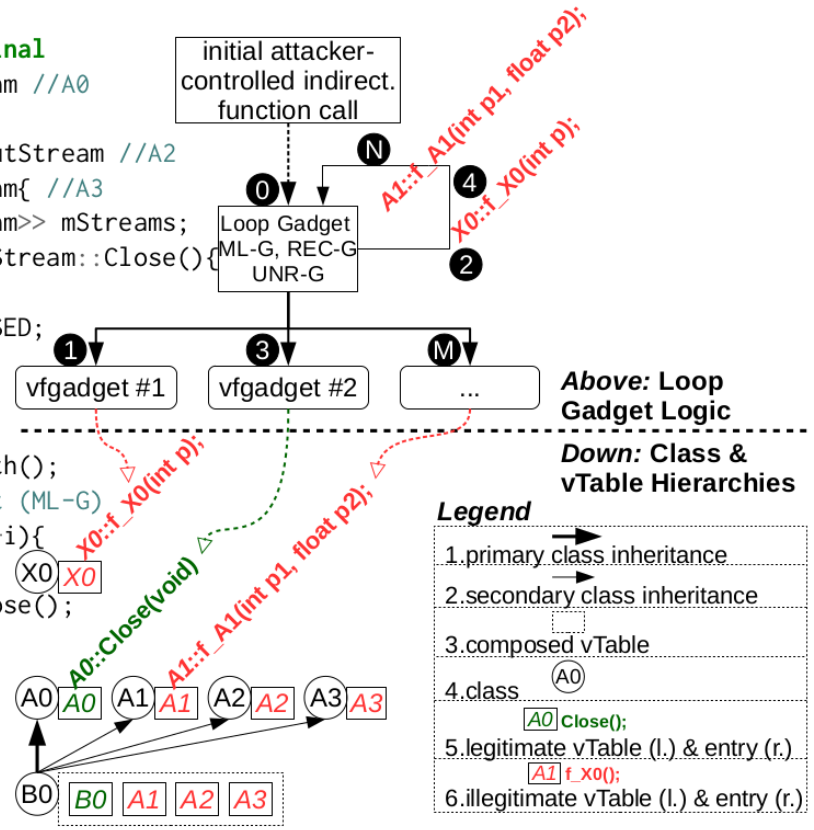

Figure 2: COOP loop gadget (ML-G, REC-G, UNR-G) at work.

Code extracted from Mozilla Firefox

Overview

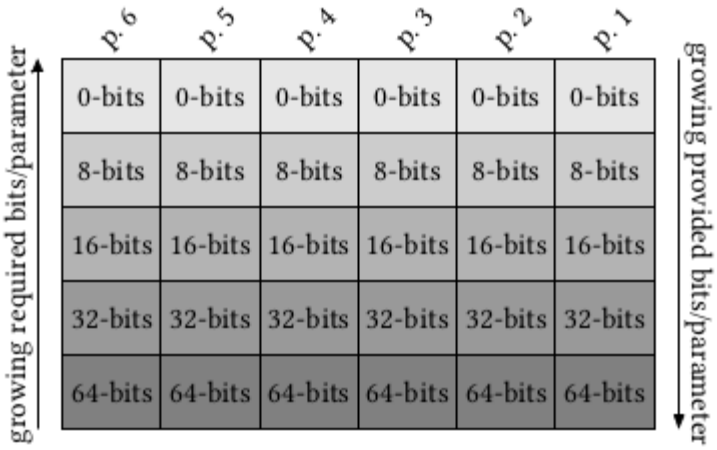

Figure 7: The *type* policy schema for callsites and calltargets. Note that p. means parameter. As it is depicted in this example, when requiring parameter width, one starts at the bottom of the above matrix and grows to the top, as it is always possible to accept more parameters than required. The reverse is true for providing parameters, as it is possible to accept less parameters than provided.

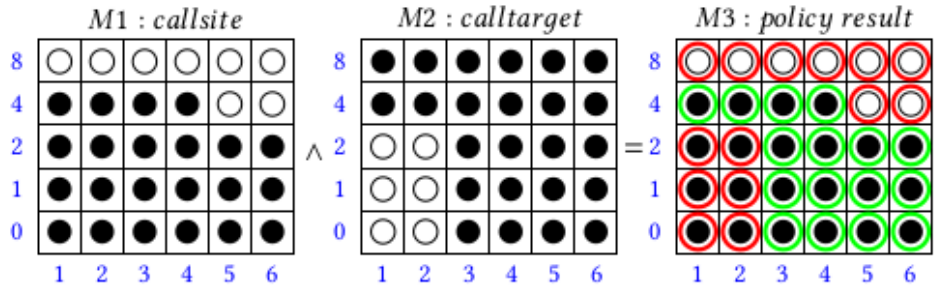

Figure 5: TypeShield's parameter type and count policy. The X and Y axis of matrices $M1, M2$ and $M3$ represent function parameter count and bit-widths in bytes, respectively. Note that our type policy performs an $\wedge$ (*i.e.*, logical and) operation between each entry in $M1_{i,j}$ and $M2_{i,j}$ where i and j are column and row indexes. If two black filled circles located in $M1 \wedge M2$ overlap on positions $M1_i = M2_i \wedge M1_j = M2_j$ than we have a match. Green circles indicate a match whereas red circles indicate a mismatch in $M3$. If at least one match is present on each of the columns of $M3$ than the indirect call transfer will be allowed by our policy, otherwise not. Note that in this example the indirect call transfer will be allowed.

Design

- Call site analysis

Algorithm 2: Basic block reaching definition analysis.

Input :The basic block to be analyzed - *block* : INSTR*
Output:The reaching definition state - $\mathcal{S}^{\mathcal{R}}$

```
1  Function analyze(block : INSTR*) : S^R is
2      state = Bl ;                          ▷ Initialize the state
3      foreach inst ∈ reversed(block) do
4          state' = analyze_instr(inst) ;    ▷ Calculate changes
5          state = merge_v(state, state') ;  ▷ Merge changes
6      end
7      states = {} ;                         ▷ Set of predecessor states
8      blocks = pred(block) ;                ▷ Get predecessors(pred.)
9      foreach block' ∈ blocks do
10         state' = analyze(block') ;        ▷ Analyze pred. block
11         states = states ∪ { state' } ;     ▷ Add pred. states
12     end
13     state' = merge_h (states) ;           ▷ Merge pred. states
14     return merge_v(state, state') ;       ▷ Merge to final state
15 end
```

Algorithm 2 is based on the reaching definition analysis presented in [28], which basically is a reverse depth-first traversal of basic blocks of a program. For customization, we rely on the implementation of several functions. $\mathcal{S}^{\mathcal{R}}$ is the set of possible register states depending on the specific reaching definition implementation of:

- `merge_v` : $\mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mapsto \mathcal{S}^{\mathcal{R}}$, which describes how to merge the current state with the following state change.
- `merge_h` : $\mathcal{P}(\mathcal{S}^{\mathcal{R}}) \mapsto \mathcal{S}^{\mathcal{R}}$, which describes how to merge a set of states resulting from several paths.
- `analyze_instr` : `INSTR` $\mapsto \mathcal{S}^{\mathcal{R}}$, which calculates the state state change that occurs due to the given instruction.
- `pred` : `INSTR`* $\mapsto \mathcal{P}$(`INSTR`*), which calculates the predecessors of the given block.

Design (cont.)

- Call target analysis

Algorithm 1: Basic block liveness analysis.

Input : The basic block to be analyzed - block : INSTR*
Output: The liveness state - $\mathcal{S}^{\mathcal{L}}$

```
1  Function analyze (block : INSTR*) : S^L is
2      state = Bl ;                          ▷ Initialize the state
3      foreach inst ∈ block do
4          state' = analyze_instr(inst) ;    ▷ Calculate changes
5          state = merge_h(state, state') ;  ▷ Merge changes
6      end
7      states = {} ;                         ▷ Set of successor states
8      blocks = succ(block) ;                ▷ Get successors(succ.)
9      foreach block' ∈ blocks do
10         state' = analyze(block') ;        ▷ Analyze succ. block
11         states = states ∪ { state' } ;     ▷ Add succ. states
12     end
13     state' = merge_h (states) ;           ▷ Merge succ. states
14     return merge_v(state, state') ;       ▷ Merge to final state
15 end
```

Algorithm 1 is based on the liveness analysis algorithm presented in [28], which basically is a depth-first traversal of basic blocks. For customization, we rely on the implementation of several functions. $\mathcal{S}^{\mathcal{L}}$ is the set of possible register states depending on the specific liveness implementation of:

- `merge_v` : $\mathcal{S}^{\mathcal{L}} \times \mathcal{S}^{\mathcal{L}} \mapsto \mathcal{S}^{\mathcal{L}}$, which describes how to merge the current state with the following state change.
- `merge_h` : $\mathcal{P}(\mathcal{S}^{\mathcal{L}}) \mapsto \mathcal{S}^{\mathcal{L}}$, which describes how to merge a set of states resulting from several paths.
- `analyze_instr` : INSTR $\mapsto \mathcal{S}^{\mathcal{L}}$, which calculates the state change that occurs due to the given instruction
- succ : INSTR* $\mapsto \mathcal{P}$(INSTR*), which calculates the successors of the given block.

Evaluation

We evaluated TypeShield by instrumenting various open source applications and conducting a thorough analysis. Our test sample includes the two ftp server applications *Vsftpd* (v.1.1.0) and *Proftpd* (v.1.3.3), the two http server applications *Postgresql* (v.9.0.10) and *Mysql* (v.5.1.65), the memory cache application *Memcached* (v.1.4.20) and the *Node.js* server application (v.0.12.5). We chose these applications, which are a subset of the applications also used in the TypeAmor paper [52], to allow for comparison. We addressed the following research questions (RQs).

- **RQ1:** How **precise** is TypeShield? (§7.1)
- **RQ2:** How **effective** is TypeShield? (§7.2)
- **RQ3:** What is the **runtime overhead** of TypeShield? (§7.3)
- **RQ4:** What is TypeShield's **instrumentation overhead**? (§7.4)
- **RQ5:** What **security level** does TypeShield offer? (§7.5)
- **RQ6:** Which **upper bounds** can TypeArmor enforce? (§7.6)
- **RQ7:** Is TypeShield **superior** compared to other tools? (§7.7)

Comparison Method. As we did not had access to the source code of TypeArmor during development of TypeShield we implemented two modes in TypeShield. The first mode of our tool is a similar implementation of the *count* policy described by TypeArmor. The second mode is our implementation of the *type* policy on top of the *count* policy implementation.

Experimental Setup. Our used setup consisted in a VirtualBox (version 5.0.26r) instance, in which we ran a Kubuntu 16.04 LTS (Linux Kernel version 4.4.0). We had access to 3GB of RAM and 4 out of 8 provided hardware threads (Intel i7-4170HQ @ 2.50 GHz).

RQ1: Precision

O2	calltargets			callsites		
Target	match	Clang miss	tool miss	match	Clang miss	tool miss
proftpd	1202	0 (0.0%)	1 (0.08%)	157	0 (0.0)	0 (0.08)
pure-ftpd	276	1 (0.36%)	0 (0.0%)	8	2 (20.0)	5 (0.0)
vsftpd	419	0 (0.0%)	0 (0.0%)	14	0 (0.0)	0 (0.0)
lighttpd	420	0 (0.0%)	0 (0.0%)	66	0 (0.0)	0 (0.0)
nginx	1035	0 (0.0%)	0 (0.0%)	269	0 (0.0)	0 (0.0)
mysqld	9952	9 (0.09%)	7 (0.07%)	8002	477 (5.62)	52 (0.07)
postgres	7079	9 (0.12%)	0 (0.0%)	635	80 (11.18)	40 (0.0)
memcached	248	0 (0.0%)	0 (0.0%)	48	0 (0.0)	0 (0.0)
node	20337	926 (4.35%)	23 (0.11%)	10502	584 (5.26)	261 (0.11)
geomean	1405.98	2.84 (0.27%)	0.93 (0.02%)	210.15	6.41 (1.8)	4.32 (0.02)

Table 1: Table shows the quality of structural matching provided by our automated verify and test environment, regarding callsites and calltargets when compiling with optimization level O2. The label Clang miss denotes elements not found in the data-set of the Clang/LLVM pass. The label tool miss denotes elements not found in the data-set of TypeShield.

O2	Calltargets			Callsites		
Target	#	perfect args	perfect return	#	perfect args	perfect return
proftpd	1009	898 (88.99%)	845 (83.74%)	157	130 (82.8%)	113 (71.97%)
pure-ftpd	128	107 (83.59%)	52 (40.62%)	8	4 (50.0%)	8 (100.0%)
vsftpd	315	270 (85.71%)	193 (61.26%)	14	14 (100.0%)	14 (100.0%)
lighttpd	289	277 (95.84%)	258 (89.27%)	66	48 (72.72%)	57 (86.36%)
nginx	913	753 (82.47%)	777 (85.1%)	269	150 (55.76%)	232 (86.24%)
mysqld	9728	7138 (73.37%)	7845 (80.64%)	8002	5244 (65.53%)	6449 (80.59%)
postgres	6873	6378 (92.79%)	5241 (76.25%)	635	500 (78.74%)	573 (90.23%)
memcached	133	123 (92.48%)	77 (57.89%)	48	47 (97.91%)	48 (100.0%)
node	20069	16853 (83.97%)	14652 (73.0%)	10502	6001 (57.14%)	8841 (84.18%)
geomean	1074.9	928.04 (86.33%)	754.09 (70.14%)	210.15	150.13 (71.43%)	185.68 (88.35%)

Table 2: The results for classification of callsites and calltargets using our *count* policy on the O2 optimization level, when comparing to the ground truth obtained by our Clang/LLVM pass. The label perfect args denotes all occurrences when our result and the ground truth perfectly match regarding the required/provided arguments. The label perfect return denotes this for return values.

RQ1: Precision (cont.)

O2	Calltargets			Callsites		
Target	#	perfect args	perfect return	#	perfect args	perfect return
proftpd	1009	835 (82.75%)	861 (85.33%)	157	125 (79.61%)	113 (71.97%)
pure-ftpd	128	101 (78.9%)	54 (42.18%)	8	4 (50.0%)	8 (100.0%)
vsftpd	315	256 (81.26%)	179 (56.82%)	14	14 (100.0%)	14 (100.0%)
lighttpd	289	253 (87.54%)	244 (84.42%)	66	48 (72.72%)	57 (86.36%)
nginx	913	639 (69.98%)	753 (82.47%)	269	140 (52.04%)	232 (86.24%)
mysqld	9728	6141 (63.12%)	7684 (78.98%)	8002	4477 (55.94%)	6449 (80.59%)
postgres	6873	5730 (83.36%)	4952 (72.05%)	635	455 (71.65%)	573 (90.23%)
memcached	133	110 (82.7%)	70 (52.63%)	48	43 (89.58%)	48 (100.0%)
node	20069	15161 (75.54%)	13911 (69.31%)	10502	4757 (45.29%)	8841 (84.18%)
geomean	1074.9	838.46 (77.99%)	726.89 (67.61%)	210.15	139.18 (66.22%)	185.68 (88.35%)

Table 3: The result for classification of callsites using our *type* policy on the O2 optimization level, when comparing to the ground truth obtained by our Clang/LLVM pass. The label perfect args denotes all occurrences when our result and the ground truth perfectly match regarding the required/provided arguments. The label perfect return denotes this for return values.

RQ2: Effectiveness

O2	AT	*count**		*count*		*type**		*type*	
Target		limit (mean ± σ)	median	limit (mean ± σ)	median	limit (mean ± σ)	median	limit (mean ± σ)	median
proftpd	396	330.31 ± 48.07	343.0	334.5 ± 51.26	311.0	310.58 ± 60.33	323.0	337.41 ± 54.09	336.0
pure-ftpd	13	5.5 ± 4.82	6.5	9.87 ± 4.32	13.0	4.37 ± 4.92	2.0	8.12 ± 4.11	7.0
vsftpd	10	7.14 ± 1.81	6.0	7.85 ± 1.39	7.0	5.42 ± 0.95	6.0	6.42 ± 0.96	7.0
lighttpd	63	27.75 ± 10.73	24.0	41.19 ± 13.22	41.0	25.1 ± 8.98	24.0	41.42 ± 14.29	38.0
mysqld	5896	2804.69 ± 1064.83	2725.0	4281.71 ± 1267.78	4403.0	2043.58 ± 1091.05	1564.0	3617.51 ± 1390.09	3792.0
postgres	2504	1964.83 ± 618.28	2124.0	1990.59 ± 574.53	2122.0	1747.22 ± 727.08	2004.0	1624.07 ± 707.58	1786.0
memcached	14	11.91 ± 2.84	14.0	12.0 ± 1.38	13.0	9.97 ± 1.45	11.0	10.25 ± 0.77	10.0
node	7230	3406.07 ± 1666.9	2705.0	5306.05 ± 1694.73	5429.0	2270.28 ± 1720.32	1707.0	4229.22 ± 2038.64	3864.0
geomean	216.61	129.77 ± 43.99	127.62	166.09 ± 40.28	171.97	105.13 ± 38.68	92.74	144.06 ± 38.38	141.82

Table 4: Restriction results of allowed callsites per calltarget for several test series on various targets compiled with Clang using optimization level O2. Note that the basic restriction to address taken only calltargets (see column AT) is present for each other series. The label *count denotes the best possible reduction using our *count* policy based on the ground truth collected by our Clang/LLVM pass, while *count* denotes the results of our implementation of the *count* policy derived from the binaries. The same applies to *type** and *type* regarding the *type* policy. A lower number of calltargets per callsite indicates better results. Note that our *type* policy is superior to the *count* policy, as it allows for a stronger reduction of allowed calltargets. We consider this a good result which further improves the state-of-the-art. Finally, we provide the median and the pair of mean and standard deviation to allow for a better comparison with other state-of-the-art tools.**

RQ3: Runtime Overhead

- We measured an average 4% runtime overhead when instrumenting with DynInst
- One reason for the performance drop includes cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching
- This is also dependent on the actual structure of the target, as it depends on the number of indirect control flow operations pertime unit
- Another reason for the slightly higher (yer acceptable) overhead is due to our runtime policy which is more complex than that of other state-of-the-art tools (3%)

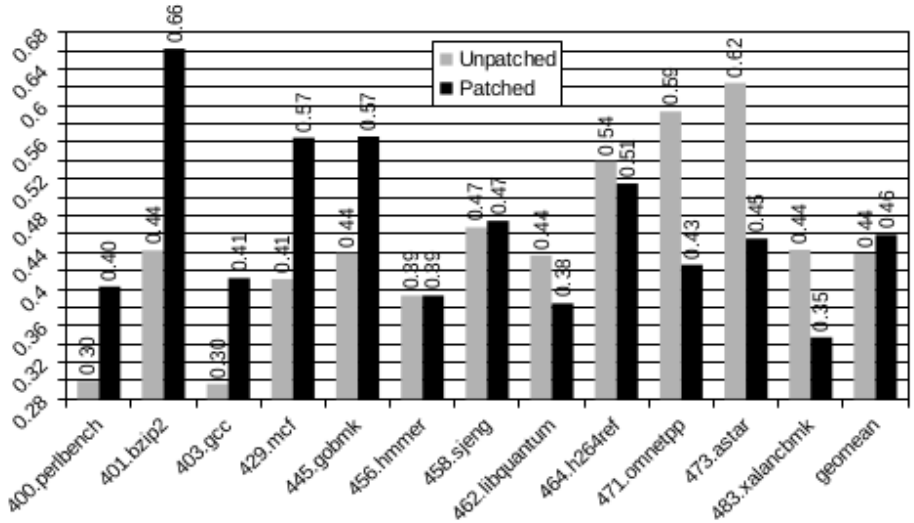

Figure 9: SPEC CPU2006 Benchmark Results.

RQ4: Instrumentation Overhead

- Binary blow up was on average 50% and up to 150% for Postgres
- This is due to relocations and the fact that each basic block in the old code contains a jump instruction to the new position of 0 the basic block
- There is still place for tweeks and we are currently addressing them

RQ5: Security Analysis of TypeShield

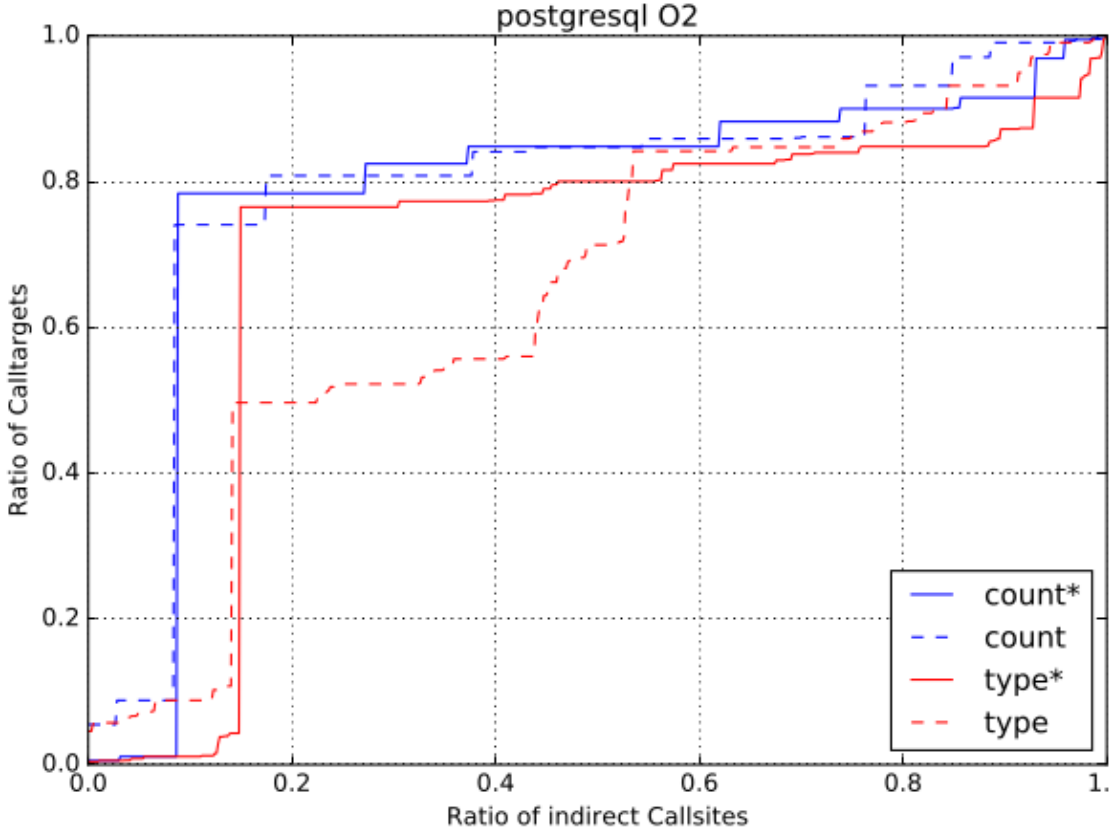

Figure 10: Postgresql -O2 CDF.

RQ6: TypeArmor Upped Bounds: Ideally

TypeArmor ***ideally*** would allow for a single callsite a set of calltargets containing a maximum of 117649 possibilities if we consider the maximum value of provided parameters to be $p = 6$ (due to $p \in [1, 6]$ possible provided parameters). Now, consider 7 C++ integer parameter types t: *int*, *char*, *unsignedchar*, *bool*, *long*, *unsignedlong*, and *short*. Thus, we obtain $t^p = 7^6 = 117649$ allowed calltargets per callsite if TypeArmor is used. Note that for simplicity reasons we considered $t = 7$ but in practice t is often even larger since there are many types of parameters in C++. The complete list of fundamental C++ types contains 20 types; not including data structures or object types. Thus, all these data types would be ignored by TypeArmor. Also, note that all other callsites having more than 6 parameters would be not checked by TypeArmor as well.

RQ6: TypeArmor Upped Bounds: Actually

TypeArmor ***actually*** allows more than t^p calltargets per callsite. If we have $t = 7$ integer types due to TypeArmors overestimation and underestimation we get for each callsite an additional number of calltargets. Let $p = 6$, then we get $c = 6x + 5y + 4z + 3t + 2p + 1v$ where: x is the sum of all calltargets consuming 6 parameters, y is the sum of all calltargets consuming 5 parameters and so on down to 0 parameters. Note that this holds since TypeArmor allows more parameters to be provided than consumed by the calltarget. Then, $c = 2100 = 600 + 500 + 400 + 300 + 200 + 100$ iff $x = y = z = t = p = v = 100$. Note that $x = 100$ is feasible under realistic conditions in large applications (*i.e.*, Google Chrome, Firefox). Next 2100 is added to 7^6. Thus, for a single callsite providing $p = 6$ parameters TypeArmor allows theoretically in total $7^6 + 2100 = 1197496$ calltargets for each callsite. Similar reasoning applies to $p = 5$ where we get $7^5 + (1500 = 500 + 400 + 300 + 200 + 100) = 18307$ iff $x = y = z = t = p = v = 100$ allowed calltarget per callsite, or $p \in [1, 4]$, too.

RQ7: Is TS superior to other forward edge tools?

Target	AT	TypeArmor	IFCC	TypeShield (count)	TypeShield (type)
lighttpd	63	47	6	41	38
Memcached	14	14	1	13	10
ProFTPD	396	376	3	311	336
Pure-FTPD	13	4	0	13	7
vsftpd	10	12	1	7	7
PostgreSQL	2504	2304	12	2122	1786
MySQL	5896	3698	150	4403	3792
Node.js	7230	4714	341	5429	3864
geomean	216.6	162.1	7.6	172.0	141.8

Table 5: Medians of calltargets per callsite for different tools.

Conclusion

- TS is a binary runtime fine-grained CFI-policy enforcing tool for forward edges
- TS was applied to real software such as web servers, FTP servers and the SPEC CPU2006 benchmark
- TS has higher precision w.r.t. the calltarget set per callsite than existing state-of-the-art tools, while maintaining a comparable runtime overhead of 4%
- TS improves parameter count based techniques by reducing the possible call-targets per callsite ratio by 35% with an overall reduction of more than 13% when comparing with similar state-of-the-art tools (e.g., TypeArmor)
- The outcome is a considerably reduced attack surface which can be further improved by tweaking our analysis
- TS can serve as a platform for other types of code reuse attacks mittigation
- **Future Work**
  - Improving the Structural Matching (improve our **analysis**)
  - Improving the Patching Schema (reduce **runtime overhead**)
  - Using Pointer and Memory Analysis (more detailed **parameter type lattice**)