Supplementary technical details of MSTest

I. Description of Test Cases

	ID	Description	Related Properties
acc	0–3	calculate pointer distance	overflow
	4–7	obtain inter-obj redzone size	overflow, compart., ASan
	8–11	obtain intra-obj redzone size	overflow, compart., ASan
	12–13	read RA	arb. read, ROP, compart.
	14–15	check existence of ASLR	arb. read, ROP, compart., ASLR
	16	obtain function address	arb. read, ROP, compart., CPI, PA
	17-18	locate RA on stack	arb. read, ROP, compart.
	19	obtain stack frame size	arb. read, ROP, compart.
	20	check existence of IBT	arb. read, CET
	21	read GOT code pointer	arb. read, ROP, CPI, RELRO
mss	22–29	intra-obj out-bound read by index	overflow, compart., ASan, tag
	30–37	intra-obj out-bound read by pointer	overflow, compart., ASan, tag
	38–45	generic array out-bound read, pinpointed overflows	overflow, compart., ASan, TBI, LAM, UAI
	46–53	long dist. intra-obj out-bound read	overflow, compart., ASan,tag
	54–59	intra-obj out-bound write by index	overflow, compart., DFI, ASan, tag
	60–65	intra-obj out-bound write by pointer	overflow, compart., DFI, ASan, tag
	66	write stack data by stack pointer overflow	overflow, stack canary, PA, backward CFI, SHSTK
	67–72	use index to implement generic array out-bound write	overflow, compart., ASan
	79–82	inter-obj out-bound read by pointer	overflow, compart., ASan
	83–85	inter-obj out-bound write by pointer	overflow, compart., ASan
	86–89	out-frame-bound read and write	overflow, compart., ASan, PA, CET
	90–93	out-page-bound read and write	overflow, compart., ASan, tag
	94–117	out-region-bound read	overflow, compart., Asan, ASLR
	118–129	out-region-bound write	overflow, compart., Asan, ASLR
	130–132	spray crossing object boundary	overflow, compart., ASan
	133	spray crossing frame boundary	overflow, compart., ASan
	134–135	spray crossing page boundary	overflow, compart., ASan
	136–139	out-bound read by type confusion from scalar to array	type, compart., ASan, ADI, MTE, CHERI
	140–143	out-bound read by scalar type confusion	type, compart., ASan, ADI, MTE, CHERI
mts	144–146	read and write by double-free on heap	zeroing, randomization
	147,149	read after free on stack before/after reclaiming	zeroing, randomization
	148	check the possibility to reallocate stack frame	randomization
	150–152	write after free on stack before/after reclaiming	zeroing, randomization
	153–154,156	read after free on heap before/after reclaiming	zeroing, randomization
	155	check the possibility to reallocate buffer on heap	randomization
	157–159	write after free on heap before/after reclaiming	zeroing, randomization
cpi	160–163	deliberately read a VTable pointer	COOP
	164–166	deliberately write a VTable pointer	COOP
	167	alter a function pointer with embedded assembly	JOP, PA
	168	arithmetic operation on function pointers	JOP, PA
	169	write GOT or PLT code pointer	arb. write, ROP, CPI, RELRO
cfi-b	170–171	ROP to the call site in callee with special function	ROP, fine-grained CFI
	172	ROP with arbitrary code snippets as gadgets	ROP, coarse-grained CFI
	173	use previously returned same function as gadgets	ROP, fine-grained CFI
	174	use previously returned overloaded function as gadgets	ROP, fine-grained CFI
	175–176	use call preceded arbitrary code snippets as gadgets	ROP, fine-grained CFI
	177–180	use call preceded arbitrary function's entry as gadgets	ROP, fine-grained CFI
	181	use long and benign code segment as gadgets	ROP, fine-grained CFI
	182	unbalance the call-return pair	ROP, CET, shadow stack
	183–186	direct code injection	DEP,ROP
cfi-f	187	hijack a call to an arbitrary location	arb. execute, ROP, coarse-grained CFI
	188	hijack a call to a wrong function	ROP,fine-grained CFI
	189	hijack a call to a but complying with static analyses	ROP, path-sensitive CFI
	190–192	forge VTables with injected functions	COOP, VTable layout, read-only
	193	forge VTables with different nums parameters function	COOP, type
	194–195	forge VTables with different types parameters function	COOP, type
	196	forge VTables with generic functions	COPP, type
	197	forge VTables with different numbers functions objects	COOP, type
	198–200	forge VTables with different types functions objects	COOP, type
	201–204	forge VTables with special relationship objects	COOP, type, class hierarchy analysis
	205	forge a VTable with the arbitrary table	COOP, type
	206	forge a VTable of a released object	COOP, revocation
	207	hijack a function pointer to its overloading function	JOP, type
	208–211	hijack a call to mismatched types of args function	JOP, type, fine-grained CFI
cfi-f	212–215	hijack function by replacing code pointers with data pointers	JOP, type
	216–219	check whether a function call can be hijacked into a data region	DEP, JOP
	220	hijiack a function pointer and jump to an arbitrary location	arb. execute, ROP, coarse-grained CFI
	221–222	jump to the return address, which is stored in the heap region	coarse-grained ASLR, JOP, ROP
	223–226	hijack a indirect jump into a data region	DEP, JOP

II. Dependency of Test Cases

Explanation for dependency with examples:

Example 1: –

No dependency is required. The test case is always tested.

Example 2: 3

The test case is tested if test case 3 returns 0 (exploitable).

Example 3: 0 & 8

The test case is tested if both test cases 8 and 0 return 0.

Example 4: 0 — 8

The test case is tested if either test case 8 or 0 returns 0.

Example 5: 0 & 8, 3

*The test case is tested in two possible scenarios, **0 & 8** and **3**. Scenario **0 & 8** is checked before scenario **3**. Depending on the enabling scenario, the test case may use different macro and runtime arguments.*

Example 6: 0 & 8, –

*The test case is tested in two possible scenarios, **0 & 8** and –, where the latter is a backup scenario. The test case is always tested but may use different macro and runtime arguments in different scenarios.*

A. Generic memory access capability (`acc`)

ID	Name	Dependency
0	check-data-pointer-arithmetic-stack	–
1	check-data-pointer-arithmetic-heap	–
2	check-data-pointer-arithmetic-data	–
3	check-data-pointer-arithmetic-rodata	–
4	check-inter-obj-stack-redzone	–
5	check-inter-obj-heap-redzone	–
6	check-inter-obj-data-redzone	–
7	check-inter-obj-rodata-redzone	–
8	check-intra-obj-stack-redzone	–
9	check-intra-obj-heap-redzone	–
10	check-intra-obj-data-redzone	–
11	check-intra-obj-rodata-redzone	–
12	copy-stackra-to-heap-explicit-arith	187
13	copy-stackra-to-heap-implicit-arith	–
14	check-prog-ASLR	186
15	check-stack-region-ASLR	14
16	read-func	–
17	get-ra-offset-v-p-g0	–
18	get-ra-offset-v-p-g1	–
19	get-frame-size	–
20	check-IBT	186
21	read-GOT	20

B. Memory spatial safety (`mss`)

ID	Name	Dependency
22	read-by-enclosing-array-index-stack-overflow	8, 0, –
23	read-by-enclosing-array-index-stack-underflow	8, 0, –
24	read-by-enclosing-array-index-heap-overflow	9, 1, –
25	read-by-enclosing-array-index-data-overflow	10, 2, –
26	read-by-enclosing-array-index-rodata-overflow	11, 3, –
27	read-by-enclosing-array-index-heap-underflow	9, 1, –
28	read-by-enclosing-array-index-data-underflow	10, 2, –
29	read-by-enclosing-array-index-rodata-underflow	11, 3, –
30	read-by-enclosing-array-pointer-stack-overflow	46, –, –
31	read-by-enclosing-array-pointer-heap-overflow	47, –, –
32	read-by-enclosing-array-pointer-data-overflow	48, –, –
33	read-by-enclosing-array-pointer-rodata-overflow	49, –, –
34	read-by-enclosing-array-pointer-stack-underflow	50, –, –
35	read-by-enclosing-array-pointer-heap-underflow	51, –, –
36	read-by-enclosing-array-pointer-data-underflow	52, –, –
37	read-by-enclosing-array-pointer-rodata-underflow	53, –, –
38	read-by-bare-array-pointer-stack-overflow	4, 0, –
39	read-by-bare-array-pointer-heap-overflow	5, 1, –
40	read-by-bare-array-pointer-data-overflow	6, 2, –
41	read-by-bare-array-pointer-rodata-overflow	7, 3, –
42	read-by-bare-array-pointer-stack-underflow	4, 0, –
43	read-by-bare-array-pointer-heap-underflow	5, 1, –
44	read-by-bare-array-pointer-data-underflow	6, 2, –
45	read-by-bare-array-pointer-rodata-underflow	7, 3, –
46	read-by-enclosing-array-pointer-large-count-stack-overflow	8 & 0, –
47	read-by-enclosing-array-pointer-large-count-heap-overflow	9 & 1, –
48	read-by-enclosing-array-pointer-large-count-data-overflow	10 & 2, –
49	read-by-enclosing-array-pointer-large-count-rodata-overflow	11 & 3, –
50	read-by-enclosing-array-pointer-large-count-stack-underflow	8 & 0

Following the previous table.

ID	Name	Dependency
51	read-by-enclosing-array-pointer-large-count-heap-underflow	9 & 1, –
52	read-by-enclosing-array-pointer-large-count-data-underflow	& 2, –
53	read-by-enclosing-array-pointer-large-count-rodata-underflow	& 3, –
54	write-by-enclosing-array-index-stack-overflow	8, 0, –
55	write-by-enclosing-array-index-heap-overflow	9, 1, –
56	write-by-enclosing-array-index-data-overflow	10, 2, –
57	write-by-enclosing-array-index-stack-underflow	8, 0, –
58	write-by-enclosing-array-index-heap-underflow	9, 1, –
59	write-by-enclosing-array-index-data-underflow	10, 2, –
60	write-by-enclosing-array-pointer-stack-overflow	73, –, –
61	write-by-enclosing-array-pointer-heap-overflow	74, –, –
62	write-by-enclosing-array-pointer-data-overflow	75, –, –
63	write-by-enclosing-array-pointer-stack-underflow	76, –, –
64	write-by-enclosing-array-pointer-heap-underflow	77, –, –
65	write-by-enclosing-array-pointer-data-underflow	78, –, –
66	write-by-stack-pointer	–
67	write-by-bare-array-pointer-stack-overflow	4, 0, –
68	write-by-bare-array-pointer-heap-overflow	5, 1, –
69	write-by-bare-array-pointer-data-overflow	6, 2, –
70	write-by-bare-array-pointer-stack-underflow	4, 0, –
71	write-by-bare-array-pointer-heap-underflow	5, 1, –
72	write-by-bare-array-pointer-data-underflow	6, 2, –
73	write-by-enclosing-array-pointer-large-count-stack-overflow	8, 0
74	write-by-enclosing-array-pointer-large-count-heap-overflow	9, 1
75	write-by-enclosing-array-pointer-large-count-data-overflow	10, 2
76	write-by-enclosing-array-pointer-large-count-stack-underflow	8, 0
77	write-by-enclosing-array-pointer-large-count-heap-underflow	9, 1
78	write-by-enclosing-array-pointer-large-count-data-underflow	10, 2
79	read-cross-object-ptr-stack	38, 42
80	read-cross-object-ptr-heap	31, 35
81	read-cross-object-ptr-data	32, 36
82	read-cross-object-ptr-rodata	33, 37
83	write-cross-object-ptr-stack-overflow	67
84	write-cross-object-ptr-heap-overflow	61
85	write-cross-object-ptr-data-overflow	62
86	read-cross-frame-index	22
87	read-cross-frame-ptr	79
88	write-cross-frame-index	54 — 57
89	write-cross-frame-ptr	83
90	read-cross-page-index-stack	22
91	read-cross-page-ptr-stack	79
92	write-cross-page-index-stack	54 & 57
93	write-cross-page-ptr-stack	83
94	read-cross-segment-stack-to-heap-index	90
95	read-cross-segment-stack-to-heap-ptr	91
96	read-cross-segment-stack-to-data-index	90
97	read-cross-segment-stack-to-data-ptr	91
98	read-cross-segment-stack-to-rodata-index	90
99	read-cross-segment-stack-to-rodata-ptr	91
100	read-cross-segment-heap-to-stack-index	24 — 27
101	read-cross-segment-heap-to-stack-ptr	80
102	read-cross-segment-heap-to-data-index	24 — 27
103	read-cross-segment-heap-to-data-ptr	80
104	read-cross-segment-heap-to-rodata-index	24 — 27
105	read-cross-segment-heap-to-rodata-ptr	80
106	read-cross-segment-data-to-stack-index	25 — 28
107	read-cross-segment-data-to-stack-ptr	81
108	read-cross-segment-data-to-heap-index	25 — 28
109	read-cross-segment-data-to-heap-ptr	81
110	read-cross-segment-data-to-rodata-index	25 — 28
111	read-cross-segment-data-to-rodata-ptr	81
112	read-cross-segment-rodata-to-stack-index	26 — 29
113	read-cross-segment-rodata-to-stack-ptr	82
114	read-cross-segment-rodata-to-heap-index	26 — 29
115	read-cross-segment-rodata-to-heap-ptr	82
116	read-cross-segment-rodata-to-data-index	26 — 29
117	read-cross-segment-rodata-to-data-ptr	82
118	write-cross-segment-stack-to-heap-index	92
119	write-cross-segment-stack-to-heap-ptr	93
120	write-cross-segment-stack-to-data-index	92
121	write-cross-segment-stack-to-data-ptr	93
122	write-cross-segment-heap-to-stack-index	58, 55
123	write-cross-segment-heap-to-stack-ptr	84, 84
124	write-cross-segment-heap-to-data-index	58, 55
125	write-cross-segment-heap-to-data-ptr	84, 84
126	write-cross-segment-data-to-stack-index	59, 56
127	write-cross-segment-data-to-stack-ptr	85, 85
128	write-cross-segment-data-to-heap-index	59, 56
129	write-cross-segment-data-to-heap-ptr	85, 85

Following the previous table.

ID	Name	Dependency
130	write-spray-cross-object-stack	83
131	write-spray-cross-object-heap	84 — 84
132	write-spray-cross-object-data	85 — 85
133	write-spray-cross-frame	89
134	write-spray-cross-page-in-stack	93
135	write-spray-cross-page-in-heap	131
136	read-scalar-cast-to-array-stack-overflow	8
137	read-scalar-cast-to-array-heap-overflow	9
138	read-scalar-cast-to-array-data-overflow	10
139	read-scalar-cast-to-array-rodata-overflow	11
140	read-scalar-cast-to-scalar-stack-overflow	8
141	read-scalar-cast-to-scalar-heap-overflow	9
142	read-scalar-cast-to-scalar-data-overflow	10
143	read-scalar-cast-to-scalar-rodata-overflow	11

C. Memory temporal safety (`mts`)

ID	Name	Dependency
144	double-free	155
145	write-by-double-free-reallocate	144
146	access-by-double-free-reallocate	144
147	access-after-free-alias-stack	148
148	reallocate-stack	–
149	access-after-reclaim-stack	148
150	write-after-free-stack	–
151	write-before-reclaim-stack	150 — 148
152	write-after-reclaim-stack	148
153	access-after-free-org-heap	155
154	access-after-free-alias-heap	155 — 148
155	reallocate-heap	–
156	access-after-reclaim-heap	155
157	write-after-free-heap	–
158	write-before-reclaim-heap	157 — 155
159	write-after-reclaim-heap	155

D. Code pointer integrity (`cpi`)

ID	Name	Dependency
160	read-stack-vtable-pointer	167
161	read-heap-vtable-pointer	167
162	read-data-vtable-pointer	163
163	read-rodata-vtable-pointer	–
164	write-stack-vtable-pointer	–
165	write-heap-vtable-pointer	–
166	write-data-vtable-pointer	–
167	func-pointer-assign	–
168	func-pointer-arithmetic	167 & (0 — 1 — 2 — 3)
169	modify-GOT	21

E. Backward control-flow Integrity (`cfi-b`)

ID	Name	Dependency
170	cfi-return-to-parent-non-call-site-by-asmfunc	–
171	cfi-return-to-parent-non-call-site-by-vfunc	–
172	cfi-return-to-parent-non-call-site	176 & 170
173	cfi-return-to-parent-same-call-site	73——((18 — 17) & 66)
174	cfi-return-to-parent-same-call-site-diffargs	73——((18 — 17) & 66)
175	cfi-return-to-parent-wrong-call-site-fakefunc-offset	173
176	cfi-return-to-parent-wrong-call-site-asm-offset	173
177	cfi-return-to-peer-asm-func	–
178	cfi-return-to-peer-func	177
179	cfi-return-to-peer-mfunc	177
180	cfi-return-to-peer-vfunc	177
181	cfi-return-to-libc	178
182	cfi-return-without-call	19 & 167
183	cfi-return-to-instruction-in-rodata	172 & 18 & 223
184	cfi-return-to-instruction-in-data	172 & 18 & 224
185	cfi-return-to-instruction-in-stack	172 & 18 & 225
186	cfi-return-to-instruction-in-heap	172 & 18 & 226

F. *Forward control-flow integrity (`cfi-f`)*

ID	Name	Dependency
187	cfi-call-mid-func	189,168 & 16
188	cfi-call-wrong-func-within-static-analysis	16 — 167
189	cfi-call-wrong-func	188
190	cfi-call-fake-vtable-with-func-stack	196
191	cfi-call-fake-vtable-with-func-heap	196
192	cfi-call-fake-vtable-with-func-data	196
193	cfi-call-fake-vtable-arg-num	196
194	cfi-call-fake-vtable-arg-type	196
195	cfi-call-fake-vtable-arg-type-modified	194
196	cfi-call-fake-vtable	165 & 161, 164 & 160, 166 & 162
197	cfi-call-wrong-vtable-func-num	205
198	cfi-call-wrong-vtable-arg-num	205
199	cfi-call-wrong-vtable-arg-type	205
200	cfi-call-wrong-vtable-arg-type-modified	199
201	cfi-call-wrong-vtable-parent	203
202	cfi-call-wrong-vtable-sibling	203
203	cfi-call-wrong-vtable-child	165 & 161, 164 & 160, 166 & 162
204	cfi-call-wrong-vtable-offset	205 — 168
205	cfi-call-wrong-vtable	203
206	cfi-call-wrong-vtable-released	205 — 153 — 150
207	cfi-call-wrong-num-arg-func	198 — 189
208	cfi-call-wrong-type-arg-int2double-func	199 — 189
209	cfi-call-wrong-type-arg-op2doublep-func	199 — 189
210	cfi-call-wrong-type-arg-op2intp-func	199 — 189
211	cfi-call-wrong-type-arg-fp2dp-func	199 — 189
212	cfi-call-wrong-type-arg-dp2fp-func-rodata	211
213	cfi-call-wrong-type-arg-dp2fp-func-data	211 — 208 — 209 — 210
214	cfi-call-wrong-type-arg-dp2fp-func-stack	211
215	cfi-call-wrong-type-arg-dp2fp-func-heap	211
216	cfi-call-instruction-in-rodata	223
217	cfi-call-instruction-in-data	224 & 192
218	cfi-call-instruction-in-stack	225 & 190
219	cfi-call-instruction-in-heap	226 & 191
220	cfi-jump-mid-func	16 — 167
221	cfi-jump-func-ra-from-heap-memcpy-explicit-arith	12 — 226
222	cfi-jump-func-ra-from-heap-memcpy-implicit-arith	13 — 226
223	cfi-jump-instruction-in-rodata	220 & 212
224	cfi-jump-instruction-in-data	220 & 213
225	cfi-jump-instruction-in-stack	220 & 214
226	cfi-jump-instruction-in-heap	220 & 215

III. List of Assembly Macros and Functions

- `GET_DISTANCE(dis, pa, pb)`: Return the distance between pointer `pa` and `pb` by `dis`.
- `READ_STACK_DAT(dat, offset)` : Return the stack data `[SP+offset]` by `dat`.
- `READ_STACK_DAT_IMM(dat, offset)`: Return the stack data `[SP+offset]` by `dat`. (`offset` is an immediate number)
- `GET_RA_ADDR(ra_addr)`: Return the default location of RA of the current stack frame.
- `MOD_STACK_DAT(dat, offset)`: Revise stack data `[SP+offset]` to `dat`.
- `SET_MEM(ptr, var)`: Revise memory data `[ptr]` to `var`.
- `JMP_DAT(ptr)`: Jump to `ptr`.
- `JMP_DAT_PTR(ptr)`: Jump to `[ptr]`.
- `PASS_INT_ARG0_IMM(arg)`: Set the first numeric argument for the next function call to `arg` according to ABI.
- `PUSH_FAKE_RET(ra, fsize)`: Allocate a fake stack frame of `fsize*8` bytes with a fake RA.
- `FUNC_MACHINE_CODE`: A snippet of machine code embedded with an illegal instruction.
- `GET_SP_LOC(loc)`: Return SP by `loc`.
- `get_got_func(void **gotp, void *label, int cet)`: Return function `label()`'s location in GOT by `gotp`. Intel CET is effective when `cet` is true.

IV. Porting MSTest to Windows 11

Porting MSTest to Windows's MSVC has encountered two major obstacles: One is the missing support for embedded assembly and some GCC defined intrinsics, such as the && operator; the other is the lack of a fully POSIX compatible signal handling mechanism.

Although MSVC prohibits embedded assembly, it provides a rich set of compiler defined intrinsics. Whenever a macro of PALib shares a similar semantic to one MSVC intrinsic, it is replaced directly or with a small modification. Otherwise, we implement the same functionality using independent assembly files and link them into the final executable. However, we are forced to implement a macro into an assembly-implemented C/C++ ABI-compatible function in the independent assembly file. Let us take `MOD_STACK_DAT()` for example, which is a crucial primitive utilized by a number of ROP test cases. As shown in Listing 1, the original POSIX assembly simply stores `dat` to the memory location `[SP + offset]`. In the MSVC's version, we have to explicitly obtain the value of SP using the MSVC defined intrinsic `RtlCaptureContext()`, add it up with the `offset`, cast it into a pointer and make the final memory write. In other words, the SP-indexed write in POSIX is implemented into a write by a generic pointer.

Implementing GCC's && operator for MSVC is another interesting case. This && operator is utilized in multiple test cases to fetch the address of a label. Unfortunately, this operator is not supported by MSVC. We choose to provide a work-around by asking the scheduler to conduct a binary analysis at runtime, as shown in Fig. 1. It is done in three steps: Step 1, a script is provided to the scheduler to retrieve the offset of the targeted label against the entry of the declaring function at runtime. Step 2, this offset is then fed to the test case as a runtime argument. Finally, step 3, the test case figures out the address of the targeted label by adding the offset and the function pointer, which must be retrieved by the test case itself due to potential ASLR.

Although the Windows exception handling does not strictly follow the POSIX standard, the semantics are similar. We have managed to achieve the same functionality with a minimal modification upon the Windows kernel's SEH (Structured Exception Handling). We need to map the Windows's exception numbers to POSIX's signal numbers as they are different: e.g. exception `EXCEPTION_ACCESS_VIOLATION` is equivalent to `SIGSEGV` in POSIX. MSTest also utilize the exception handling to verify that an exception is indeed raised due to certain defense by checking the memory address, signal type, etc. In MSVC's implementation, such checks are done at the outermost level while the crucial information needed for the check may have been lost in the inner levels due to the embedded exception handling procedure (information loss when an exception is thrown to the outer catcher). To improve the accuracy of exception detection, we set the exception flag captured by the inner function to `EXCEPTION_CONSTINUE_SEARCH` to maintain the conciseness of the current function. Therefore, the exception checks at the outermost level is still possible to do a detailed analysis.

TABLE I
COMPARISON OF REDUCING TIME BY MSTEST.

	Exhausted-Run Time (s)	Fast-Run Time (s)	Time Reduction	Cases Skipped	Accuracy Accuracy
Armv8.6-Dar-L	25.8	25.6	0.8%	10	100.00%
LLVM-ASan-none	22.5	22.3	1.1%	8	100.00%
Armv8.4-Lin-G	33.3	31.7	4.8%	9	100.00%
G8-LLVM-full	46.4	42.9	7.5%	9	100.00%
M2-Apple-PA	26.6	24.1	9.5%	34	100.00%
G8-GCC-full	54.7	48.0	12%	18	100.00%
X64-LLVM-full	28.0	24.4	13%	12	100.00%
X64-GCC-full	31.3	25.6	18%	21	100.00%
X64-Win-full	187	142	24%	18	100.00%
Morello-default	3318	2403	28%	87	96.55%
G8-Win-full	351	253	28%	15	100.00%
Cheri-default	3362	2405	28%	78	98.72%
G3-LLVM-ASan-full	666	468	30%	70	95.71%
G3-GCC-ASan-full	896	621	31%	62	95.16%
Morello-strong	3411	2097	39%	97	97.94%
x86-LLVM-ASan-full	35.0	20.8	41%	63	100.00%
M2-ASan-full	53.1	31.2	41%	70	97.14%
Cheri-strong	3394	1988	41%	93	96.77%
G8-Win-none	429	250	42%	12	100.00%

V. REDUCING TESTING TIME WITH FAST-RUN

RecIPE is the most relevant test suite with MSTest. Running the 204 test cases of RecIPE on platform x86-64-Lin-G takes 29.3 and 30.3 seconds for compilation and execution, respectively. It is 21.7 and 0.6 seconds, respectively, for running the 227

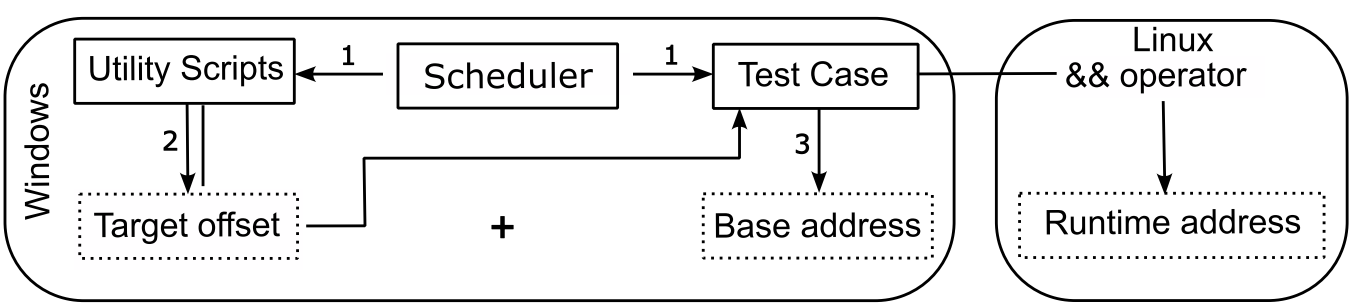

Fig. 1. Implementing GCC's && operator in MSVC.

Listing 1
PORT MOD_STACK_DAT() TO MSVC.

```
//POSIX version
#define MOD_STACK_DAT(dat, offset)              \
  asm volatile("movq %1, %%rcx;"                \
  "addq %%rsp, %%rcx;" "movq %0, (%%rcx);"      \
    :: "r"(dat), "r"(offset): "rcx")

//MSVC version
#define MOD_STACK_DAT(dat, offset)              \
  RtlCaptureContext(&sp_loc_context);           \
  offset += (long long)sp_loc_context.Rsp ;     \
  void** ptr = (void**)offset; *ptr = dat
```

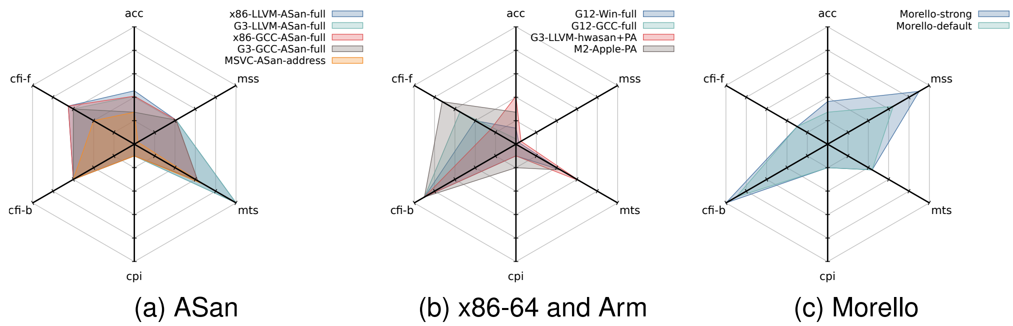

Fig. 2. Graphic representation for the protection of memory safety using different defense combinations.

test cases of MSTest, reaching a wider coverage with 63% less time. The relation graph is utilized in the fast-run mode to automatically resolve dependency between test cases. On platform x86-64-Lin-G, the exhausted-run takes 21.7 and 0.8 seconds for compilation and execution, respectively. The fast-run skips 11 test cases (4.6%) and reduces both time by 0.2% and 23%, respectively, Although the number of skipped test cases is small, the reduction in execution time is substantial as skipping test cases is not the only source of time reduction. If we recall the dependency described in Section II, without a proper order resolved by the relation graph, test cases like `return-to-wrong-call-site` fall back to retries, which cost extra time. When defenses are stronger, more test cases can be skipped in the fast-run. On Morello-strong, the exhausted run takes 40.7 and 4.0 minutes for compilation and execution, respectively. The time is long as Morello is emulated by Arm FVP. The fast-run skips 106 test cases (44%) and significantly reduce the time for compilation and execution by 38% and 64%, respectively.

VI. Effectiveness of ASans and defense combinations on different platforms

To better present the protection of memory safety using different defense combinations on various platforms, we use spider chart to analyze the defense coverage of each defense combination using the six categories of test cases (`acc`, `mss`, `mts`, `cpi`, `cfi-b` and `cfi-f`) as dimensions. On each dimension, the wider coverage denotes better protection as more test cases belonging to this category has been prevented. The result is illustrated in Fig. 2. Fig. 2a depicts the result of deploying various sanitizers. As shown in Fig. 2b, the memory safety provided by different non-sanitizer defense combinations on different architectures (x86-64, Arm and Apple) is complicated. Among these mitigation, Apple's PA provides the best CFI protection. Finally, Fig. 2c shows that Morello achieves the strongest protection against out-of-bound accesses (`mss`) and ROP attacks (`cfi-b`) while its protection for UAF on heap and access capability is not bad either.

VII. Comparison with RecIPE

Given that the RecIPE exclusively accommodates the x64-Linux platform and lacks temporary security test cases, we will restrict our comparison to the defense mechanisms deployed on x64-Linux to ensure fairness. Since the $\_free\_hook$ mechanism was removed after glibc 2.34 and the Intel CET mechanism was only supported starting from glibc 2.39, we have removed 20 tests in the RecIPE that used the $\_free\_hook$ library function. All test cases of the RecIPE for Intel CET failed during exploitation, and in our test, the case utilized embedded assembly attack primitives and the gadget with `notrack` prefix has successfully bypassed it. Compared to the RecIPE's default test results, ours do not show inconsistencies in memory spatial safety false positives between GCC and LLVM. In addition, due to the limited coverage of the RecIPE's test cases, there is no detection of the differences between GCC and LLVM in implementing RELRO. In summary, RecIPE lacks fine-grained differentiation in attack primitives for certain defensive mechanisms, making it unsuitable for cross-platform evaluation.

VIII. The configuration of full-defense tests on commercially available platforms.

TABLE II
The configuration of all defenses on commercially available platforms.

Abbr.	Extra Compiler Flags
i712-Linux	`-Wstack-protector -fstack-protector-all -mstack-protector-guard=global -fsanitize=cfi -fvisibility=hidden -flto -fPIE -fcf-protection=full -fstack-clash-protection -fsanitize=address -fsanitize-address-use-after-scope -fno-common -fsanitize=pointer-compare -fsanitize=pointer-subtract -fno-sanitize-recover=all -U_FORTIFY_SOURCE -fsanitize-address-use-after-return=always -fsanitize=undefined`
i512-Windows	`/guard:cf /sdl /GS /RTCs /fsanitize=address /link /incremental:no /OPT:NOREF /OPT:NOICF /GUARD:CF /CETCOMPAT`
i712-OpenBSD	`-Wstack-protector -fstack-protector-all -mstack-protector-guard=global -fstack-clash-protection`
G4-Linux	`-march=armv8.5-a+pauth -Wstack-protector -fstack-protector-all -fstack-clash-protection -fsanitize=address -fsanitize-address-use-after-scope -fno-common -fsanitize=pointer-compare -fsanitize=pointer-subtract -D_FORTIFY_SOURCE=2 -fsanitize-address-use-after-return=always -fsanitize=undefined -mbranch-protection=pac-ret -mbranch-protection=bti`
M2-MacOS	`-arch arm64e -Wall -Wstack-protector -fstack-protector-all -fstack-clash-protection -fsanitize=address -fsanitize-address-use-after-scope -fno-common -fsanitize=pointer-compare -fsanitize=pointer-subtract -D_FORTIFY_SOURCE=2 -fsanitize-address-use-after-return=always -fsanitize=undefined -D_FORTIFY_SOURCE=2`
TG3-Android	`-march=armv8.5-a+memtag+pauth -Wall -Wstack-protector -fstack-protector-all -fsanitize=cfi -fvisibility=hidden -fstack-clash-protection -fsanitize=address -fsanitize-address-use-after-scope -fno-common -fsanitize=pointer-compare -fsanitize=pointer-subtract -D_FORTIFY_SOURCE=2 -fsanitize=undefined -fsanitize-address-use-after-return=always -mbranch-protection=standard -fcf-protection=all`
Morello-CheriBSD	`Wstack-protector -fstack-protector-all -fstack-clash-protection -fsanitize-address-use-after-scope -fsanitize-address-use-after-return=always -march=morello -mabi=purecap -cheri-bounds=everywhere-unsafe`