Chronomorphic Programs: Using Runtime Diversity to Prevent Code Reuse Attacks

Scott E. Friedman, David J. Musliner, and Peter K. Keller
Smart Information Flow Technologies (SIFT)
Minneapolis, USA
email: {sfriedman,dmusliner,pkeller}@sift.net

Abstract—**Return Oriented Programming (ROP) attacks, in which a cyber attacker crafts an exploit from instruction sequences already contained in a running binary, have become popular and practical. While previous research has investigated software diversity and dynamic binary instrumentation for defending against ROP, many of these approaches incur large performance costs or are susceptible to Blind ROP attacks. We present a new approach that automatically rewrites potentially-vulnerable software binaries into chronomorphic binaries that change their in-memory instructions and layout repeatedly, at runtime. We describe our proof of concept implementation of this approach, discuss its security and safety properties, provide statistical analyses of runtime diversity and reduced ROP attack likelihood, and present empirical results that demonstrate the low performance overhead of actual chronomorphic binaries.**

Keywords-cyber defense; software diversity; self-modifying code.

I. INTRODUCTION

In the old days, cyber attackers only needed to find a buffer overflow or other vulnerability and use it to upload their exploit instructions, then make the program execute those new instructions. To counter this broad vulnerability, modern operating systems enforce “write XOR execute” defenses: that is, memory is marked as either writable or executable, but not both. So exploit code that is uploaded to writable memory cannot be executed. Not surprisingly, attackers then developed a more sophisticated exploit method.

Computer instruction sets are densely packed into a small number of bits, so accessing those bits in ways that a programmer did not originally intend can yield *gadgets*: groups of bits that form valid instructions that can be strung together by an attacker to execute arbitrary attack code from an otherwise harmless program [1][2]. Known as *Return Oriented Programming* (ROP), these types of cyber exploits have been effective and commonplace since the widespread deployment of W$\oplus$X defenses. Software with a single small buffer-overflow vulnerability can be hijacked into performing arbitrary computations using ROP techniques. Hackers have even developed *ROP compilers* that build the ROP exploits automatically, finding gadgets in the binary of a vulnerable target and stringing those gadgets together to implement the attacker's code [3][4].

This paper presents a fully automated approach for transforming binaries into *chronomorphic* binaries that diversify themselves during runtime, throughout their execution, to offer strong statistical defenses against code reuse exploits such as ROP and *Blind ROP* (BROP) attacks. The idea is to modify the binary so that all of the potentially-dangerous gadgets are repeatedly changing or moving, so that even a BROP attack tool cannot accumulate enough information about the program's memory layout to succeed.

In the following sections, we discuss related research in this area (Section II) and describe how our prototype Chronomorph tool converts regular binaries into chronomorphic binaries (Section III) and review its present limitations. We then describe an analysis of the safety and security of the resulting chronomorphic binaries, and performance results on early examples (Section IV). We conclude with several directions for future work, to harden the tool and broaden its applicability (Section V).

II. RELATED WORK

Various defense methods have been developed to try to foil code reuse exploits such as ROP and BROP. Some of defenses instrument binaries to change their execution semantics [5] or automatically filter program input to prevent exploits [6]; however these approaches require process-level virtual machines or active monitoring by other processes. Other approahces separate and protect exploitable data (e.g., using shadow stacks [7]), but such approaches incur comparatively high overhead.

To reduce overhead and maintain compatibility with existing operating systems and software architectures, many researchers have focused on lightweight, diversity-based techniques to prevent code reuse exploits. For example, Address Space Layout Randomization (ASLR) is common in modern operating systems, and loads program modules into different locations each time the software is started. However, ASLR does not randomize the location of the instructions *within* loaded modules, so programs are still vulnerable to ROP attacks [8]. Some diversity techniques modify the binaries themselves to make them less predictable by an attacker. For example:

- Compile-time diversity (e.g., [9]) produces semantically equivalent binaries with different structures.
- Offline code randomization (e.g., [10]) transforms a binary on disk into a functionally equivalent variant with different bytes loaded into memory.
- Load-time code randomization (e.g., [11][12]) makes the binary load blocks of instructions at randomized addresses.

These diversity-based approaches incur comparatively lower overhead than other ROP defenses and they offer statistical guarantees against ROP attacks.

Unfortunately, these compile-time, offline, and load-time diversity defenses are still susceptible to *Blind* ROP (BROP) attacks that perform runtime reconnaissance to map the binary and find gadgets [13]. So even with compile-time, offline, or load-time diversity, software that runs for a significant period of time without being reloaded (e.g., all modern server architectures) is vulnerable. Some ROP defenses modify the operating system to augment diversity [14][15] provide promising results, but these approaches do not modify existing third-party programs to work on existing operating systems.

Unlike the above diversity techniques, chronomorphic programs diversify themselves *throughout program execution* to statistically prevent code reuse attacks even if the attacker knows the memory layout.

III. Approach

The Chronomorph approach requires changing machine code at runtime, a technique known as *self-modifying code* (SMC). Using SMC, Chronomorph must preserve the functionality of the underlying program (i.e., maintain semantics), maximize diversity over time, and minimize performance costs.

Any SMC methodology requires a means to change the permissions of the program's memory (i.e., temporarily circumvent W$\oplus$X defense) to modify the code and then resume its execution. Different operating systems utilize different memory protection functions, e.g., `mprotect` in Linux and `VirtualProtect` in Windows, but otherwise, the basic instruction set architectures (e.g., x86) are equivalent. In this paper, we describe Chronomorph in a 32-bit Linux x86 setting.

Our approach automatically constructs chronomorphic binaries from normal third-party programs with the following enumerated steps, also illustrated in Figure 1:

Offline:

1) Transform the executable to inject the Chronomorph SMC runtime that invokes `mprotect` and rewrites portions of the binary during execution. This produces a *SMC binary* with SMC functions that are disconnected from the program's normal control flow.
2) Analyze the SMC binary to identify potentially-exploitable sequences of instructions (i.e., gadgets).
3) Identify relocatable gadgets and transform the SMC binary to make those gadgets relocatable.
4) Compute instruction-level, semantics-preserving transforms that denature non-relocatable gadgets and surrounding program code.
5) Write the relocations and transforms to a *morph table* outside the chronomorphic binary.
6) Inject *morph triggers* into the SMC binary so that the program will morph itself periodically. This produces the *chronomorphic binary*.

Online:

7) During program runtime, diversify the chronomorphic binary's executable memory space by relocating and transforming instructions without hindering performance or functionality.

We have implemented each step in this process and integrated third-party tools including a ROP compiler [4], the Hydan tool for computing instruction-level transforms [16], and the open-source `objdump` disassembler. We next describe each of these steps in this process, including the research challenges and the strategy we employ in our Chronomorph prototype implementation. We note relevant simplifying assumptions in our prototype, and we address some remaining research challenges in Section V.

A. Injecting SMC morphing functionality

Before the Chronomorph analysis tool can analyze the binary and compute transformations, it must inject the Chronomorph SMC runtime, which contains functions for modifying memory protection (e.g., `mprotect`), writing byte sequences to specified addresses, and reading the morph table from outside the binary. These Chronomorph functions may *themselves* contain gadgets and have runtime diversification potential, so the SMC-capable binary should be the subject of all further offline analysis.

We identified three ways of automatically injecting the Chronomorph runtime code, based on the format of the target program.

1) Link the target program's source code against the compiled Chronomorph runtime. This produces a dynamically- or statically-linked SMC executable. This is the simplest solution, and the one used in our experiments, but source code may not always be available.
2) Rewrite a statically-linked binary by extending its binary with a new loadable, executable segment containing the statically-linked Chronomorph runtime. This produces a statically-linked SMC executable.
3) Rewrite a dynamically-linked binary by adding Chronomorph procedures and objects to an alternative procedure linkage table (PLT) and global object table (GOT), respectively, and then extend the binary with a new loadable, executable segment containing the dynamically-linked Chronomorph runtime. This produces a dynamically-linked SMC executable.

All three of these approaches inject the self-modifying Chronomorph runtime, producing the *SMC binary* shown in Figure 1. At this point, the self-modification functions are not yet invoked from within the program's normal control flow, so we cannot yet call this a chronomorphic binary.

B. Identifying exploitable gadgets

As shown in Figure 1, the Chronomorph offline analysis tool includes a third-party ROP compiler [4] that automatically identifies available gadgets within a given binary and creates an exploit of the user's choice (e.g., execute an arbitrary shell command) by compiling a sequence of *attack gadgets* from the available gadgets, if possible. The Chronomorph analysis tool runs the ROP compiler against the SMC binary, finding

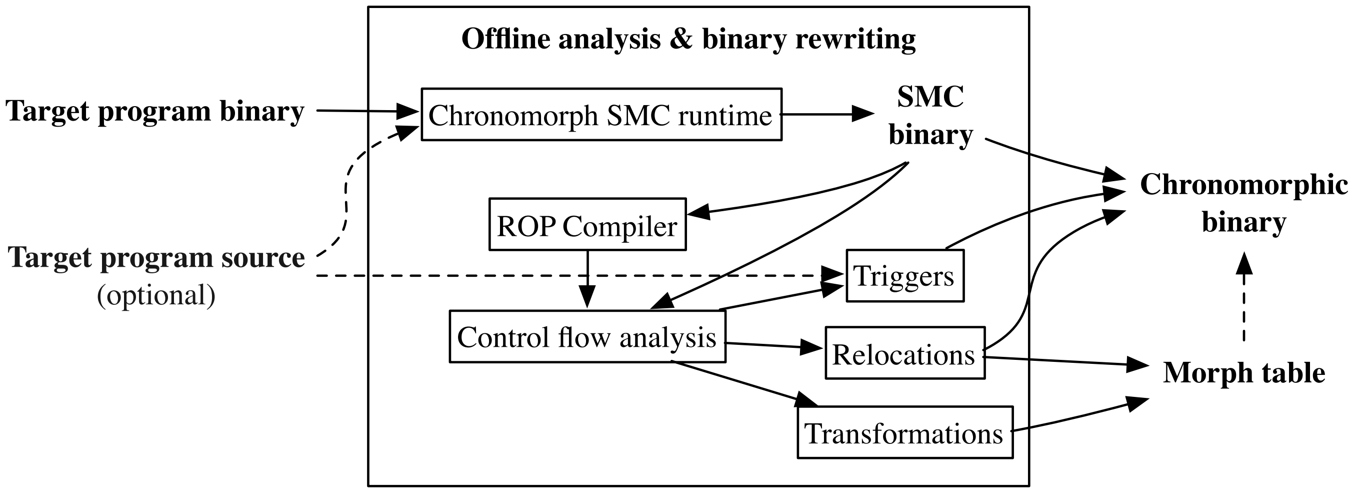
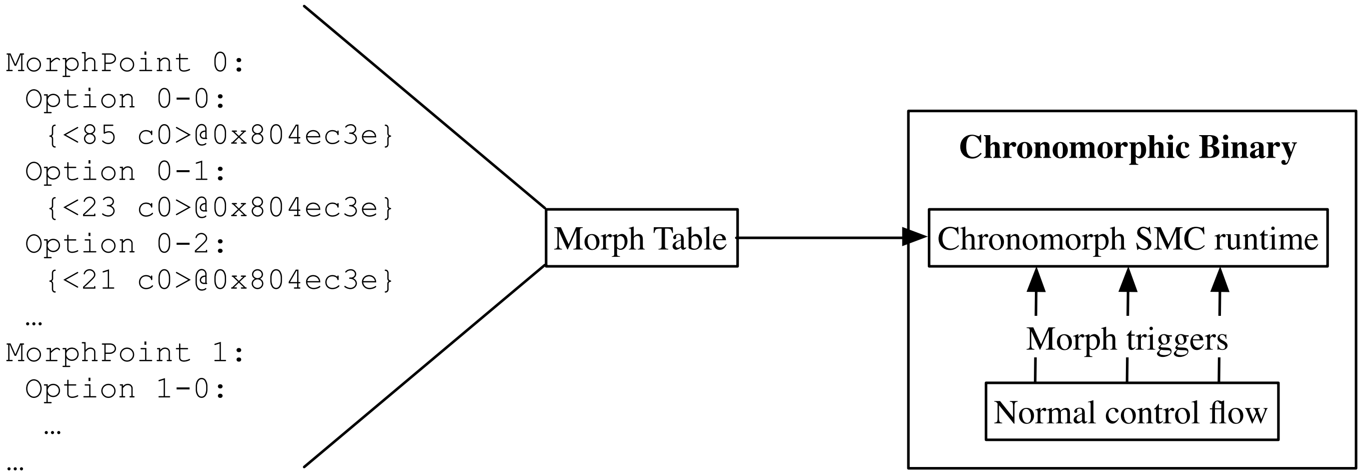


Figure 1. Chronomorph converts a third-party program into a chronomorphic binary.



```
MorphPoint 0:
 Option 0-0:
  {<85 c0>@0x804ec3e}
 Option 0-1:
  {<23 c0>@0x804ec3e}
 Option 0-2:
  {<21 c0>@0x804ec3e}
 ...
MorphPoint 1:
 Option 1-0:
  ...
...
```

Figure 2. The resulting chronomorphic binary and its interaction with the morph table.

gadgets that span the entire executable segment, including the Chronomorph SMC runtime.

The ROP compiler prioritizes Chronomorph's diversification efforts as follows, to allocate time and computing resources proportional to the various threat of exploit within the binary:

- Attack gadgets are highest priority. The chronomorphic binary should address these with highest-diversity transforms.
- Available gadgets (i.e., found by the ROP compiler but not present in an attack sequence) are medium priority. These too should be addressed by high-diversity transforms within acceptable performance bounds.
- Instructions that have not been linked to an available gadget are lowest priority, but should still be diversified (i.e., substituted or transformed in-place). Since zero-day gadgets and code-reuse attack strategies may arise after transformation time, this diversification offers additional security.

Our approach attempts all transformations possible, saving more costly transformations, e.g., dynamic block relocations, for the high-risk attack gadgets. The ROPgadget compiler [4] used by Chronomorph may be easily replaced by newer, broader ROP compilers, provided the compiler still compiles attacks and reports all available gadgets. Also, a portfolio approach may be used, running a variety of ROP compilers and merging their lists of dangerous gadgets.

C. Diversity with relocation

We may not be able to remove a high-risk gadget entirely from the executable, since its instructions may be integral to the program's execution; however, the chronomorphic binary can relocate it with high frequency throughout execution, as long as it preserves the control flow.

Relocation is the highest-diversity strategy that Chronomorph offers. Chronomorph allocates an empty *block relocation space* in the binary, reserved for gadget relocation. Whenever the chronomorphic binary triggers a morph, it shuffles relocated blocks to random locations in the block relocation space and repairs previous control flow with recomputed `jmp` instructions to the corresponding location in the block relocation space.

For each high-risk attack gadget, Chronomorph performs the following steps to make it relocatable during runtime:

1) Compute the *basic block* (i.e., sequence of instructions with exactly one entry and exit point) that contains the gadget.
2) Relocate the byte sequence of the gadget's basic block to the first empty area in the block relocation space.
3) Write a `jmp` instruction from the head of the basic block to the new address in the block relocation space.
4) Write `nop` instructions over the remainder of the gadget's previous basic block, destroying the gadgets.
5) Write the block's byte sequence and the address of the new `jmp` instruction to the morph table.

The morph table now contains enough information to place the gadget-laden block anywhere in the block relocation space and recompute the corresponding `jmp` instruction accordingly.

Intuitively, diversity of the binary increases with the size of the block relocation space. For a single gadget block g of with byte-size $|g|$, and block relocation space of size $|b|$, relocating g adds $V(g, b) = |b| - |g|$ additional program variants.

If we relocate multiple gadget blocks $G = \{g_0, ..., g_{|G|-1}\}$, the we add the following number of variants:

$$V(G, |b|) = \prod_{i=0}^{|G|-1} (|b| - \sum_{j=0}^{i} |g_j|). \tag{1}$$

The probability of guessing all of the relocated gadgets' addresses is therefore $1/V(G, |b|)$, which diminishes quickly as the block relocation space increases.

Our Chronomorph prototype has the following constraints for choosing gadget blocks for relocation:

- Relocated blocks cannot contain a `call` instruction. When a `call` instruction is executed, the subsequent instruction's address is pushed onto the stack, and if the calling block is relocated, execution would return into an arbitrary spot in the block relocation space.
- Relocated blocks must be at least the size of the `jmp` to the block relocation space, so that Chronomorph has room to write the `jmp`.
- Relocated blocks must end in an indirect control flow (e.g., `ret`) instruction; otherwise, we would have to recompute the control flow instruction at the block's tail at every relocation. Empirically, the vast majority of these blocks end in `ret`.
- Relocated gadgets cannot span two blocks.

We discuss some improvements in the conclusion of this paper for hardening Chronomorph and removing some of these constraints.

D. Diversity with in-place code randomizations

Chronomorph uses *in-place code randomization* (IPCR) strategies to randomize non-relocated instructions [10]. IPCR performs narrow-scope transformations without changing the byte-length of instruction sequences.

At present, Chronomorph use two IPCR strategies to compute transformations. The first, *instruction substitution* (IS), substitutes a single instruction for one or more alternatives. For example, comparisons can be performed in either order, `xor`'ing a register with itself is equivalent to `mov`'ing or `and`'ing zero, etc. These instructions have the same execution semantics, but they change the byte content of the instruction, so unintended control flow instructions (e.g., `0xC3 = ret`) are potentially transformed or eliminated. A single IS adds as many program variants as there are instruction alternatives.

Another IPCR strategy, *register preservation code reordering* (RPCR) reorders the `pop` instructions before every `ret` instruction of a function, and also reorders the corresponding `push` instructions at the function head to maintain symmetry. A register preservation code reordering for a single function adds as many variants as there are permutations of `push` or `pop` instructions.

Importantly, RPCR changes the layout of a function's stack frame, which may render it non-reentrant. For instance, if control flow enters the function and it preserves register values via `push`'ing, and then the chronomorphic binary runs RPCR on the function, it will likely `pop` values into unintended registers and adversely affect program functionality.

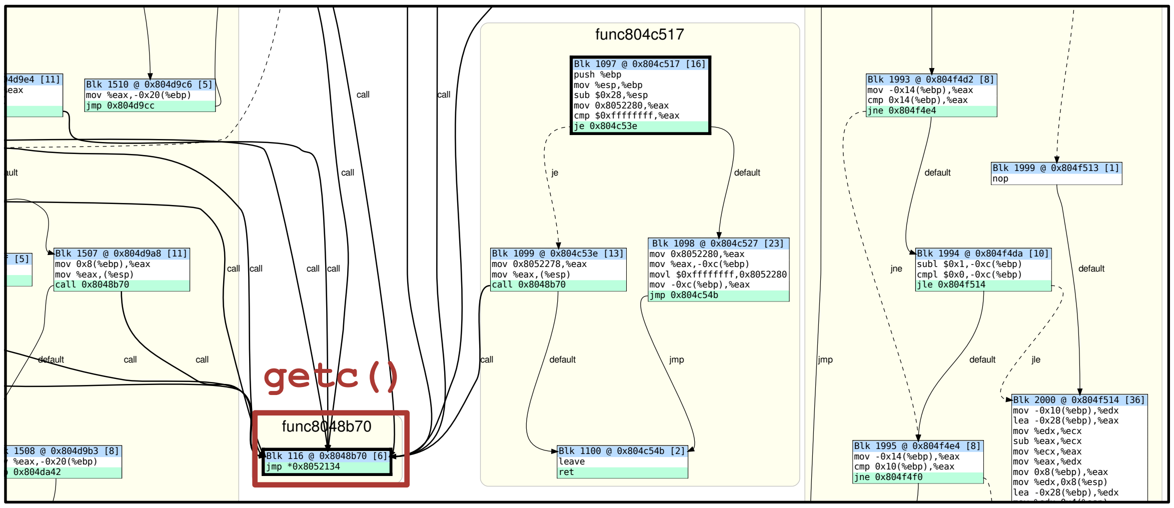
Any stack-frame diversity method such as RPCR should only be attempted at runtime if execution cannot *reenter* the function, e.g., from an internal `call`, after a SMC morph operation. We enforce this analytically with control flow graph (CFG) analysis: if execution can reenter a function *f* from the morph trigger (i.e., if the morph trigger is *reachable* from *f* in the CFG), the stack frame of *f* should not be diversified. Stack frame diversification is a valuable tool for ROP defense, but it requires these special considerations when invoked during program execution.

E. Writing and reading the morph data

The morph table is a compact binary file that accompanies the chronomorphic binary, as shown in Figure 2. The morph table binary represents packed structs: `MorphPoint` structs with internal `MorphOption` byte sequences. Each `MorphPoint` represents a decision point (i.e., an IS or RPCR opportunity) where any of the associated `MorphOption` structs will suffice. Each `MorphPoint` is stateless (i.e., does not depend on the last choice made for the `MorphPoint`), and independent of any other `MorphPoint`, so random choices are safe and ordering of the morph table is not important.

The relocation data is a separate portion of the morph table, containing the content of relocatable blocks alongside their corresponding `jmp` addresses. Like IPCR operations, relocations are stateless and independent, provided the Chronomorph runtime does not overlay them in the block relocation space.

Intuitively, the morph table cannot reside statically inside the binary as executable code, otherwise all of the gadgets would be accessible.



```
Blk 1510 @ 0x804d9c6 [5]
mov %eax,-0x20(%ebp)
jmp 0x804d9cc

Blk 1507 @ 0x804d9a8 [11]
mov 0x8(%ebp),%eax
mov %eax,(%esp)
call 0x8048b70

Blk 116 @ 0x8048b70 [6]
jmp *0x8052134

Blk 1097 @ 0x804c517 [16]
push %ebp
mov %esp,%ebp
sub $0x28,%esp
mov 0x8052280,%eax
cmp $0xffffffff,%eax
je 0x804c53e

Blk 1099 @ 0x804c53e [13]
mov 0x8052278,%eax
mov %eax,(%esp)
call 0x8048b70

Blk 1098 @ 0x804c527 [23]
mov 0x8052280,%eax
mov %eax,-0xc(%ebp)
movl $0xffffffff,0x8052280
mov -0xc(%ebp),%eax
jmp 0x804c54b

Blk 1100 @ 0x804c54b [2]
leave
ret

Blk 1993 @ 0x804f4d2 [8]
mov -0x14(%ebp),%eax
cmp 0x14(%ebp),%eax
jne 0x804f4e4

Blk 1999 @ 0x804f513 [1]
nop

Blk 1994 @ 0x804f4da [10]
subl $0x1,-0xc(%ebp)
cmpl $0x0,-0xc(%ebp)
jle 0x804f514

Blk 1995 @ 0x804f4e4 [8]
mov -0x14(%ebp),%eax
cmp 0x10(%ebp),%eax
jne 0x804f4f0

Blk 2000 @ 0x804f514 [36]
mov -0x10(%ebp),%edx
lea -0x28(%ebp),%eax
mov %edx,%ecx
sub %eax,%ecx
mov %ecx,%eax
mov %eax,%edx
mov 0x8(%ebp),%eax
mov %edx,0x8(%esp)
lea -0x28(%ebp),%edx
```

Figure 3. Small portion of a control flow graph (CFG) automatically created during Chronomorph's offline analysis. Shaded regions are functions, instruction listings are basic blocks, and edges are control flow edges.

F. Injecting morph triggers

We have described how Chronomorph injects SMC capabilities into third-party executables and its diversification capabilities, but Chronomorph must also automatically connect the Chronomorph runtime into the program's control flow to induce diversification of executable memory during runtime.

The injection of these *morph triggers* presents a trade-off: morphing too frequently will unnecessarily degrade program performance; morphing too seldom will allow wide windows of attack. Ideally, morphing will happen at the speed of input, e.g., once per server request or transaction or user input (or some modulo thereof). The location of the morph trigger(s) in the program's control flow ultimately determines morph frequency.

Figure 3 shows a portion of the CFG for the program used in our experiment, calling out the `getc()` input function. Chronomorph can inject calls to the SMC runtime at these points, or at calling functions with stack-based buffers.

Chronomorph also includes an interface for the application developer to add a specialized `MORPH` comment in the source code, which is replaced by a morph trigger during the rewriting phase.

G. Runtime diversification

A chronomorphic binary executes in the same manner as its former non-chronomorphic variant, except when the morph triggers are invoked.

When the first morph trigger is invoked, the Chronomorph runtime loads the morph binary and seeds its random number generator. All morph triggers induce a complete SMC diversification of the in-process executable memory according to IPCR and relocation data in the morph table:

1) The block relocation space is made writable with `mprotect`.
2) The block relocation space is entirely overwritten with `nop` instructions.
3) Each relocatable block is inserted to a random block relocation space address, and its `jmp` instruction is rewritten accordingly.
4) The block relocation space is made executable.
5) Each `MorphPoint` is traversed, and a corresponding `MorphOption` is chosen at random and written. Each operation is surrounded by `mprotect` calls to make the corresponding page writable and then executable. Future work will group `MorphPoints` by their address to reduce `mprotect` invocations, but our results demonstrate that the existing performance is acceptable.

We conducted an experiment with our Chronomorph prototype on a third-party Linux binary to characterize the diversity, ROP attack likelihood, and performance overhead of our Chronomorph approach. We discuss this experiment and its results in the next section.

IV. EVALUATION

We tested our prototype tool on small target binaries of Linux desktop applications, into which we deliberately injected vulnerabilities and gadgets. Here, we discuss results for the `dc` (desktop calculator) program.

The original target binary, with injected flaws, is easily compromised by our ROP compiler. After running the prototype Chronomorph system, the new binary operates as described in Figure 2, and cannot be defeated by the ROP compiler. The dynamically-linked version of the target binary is small (47KB), and after our tool has made it chronomorphic (with a block relocation space of 4KB) it is 62KB.

The rewritten binary is currently able to perform approximately 1000 changes to its own code in less than one millisecond. When the chronomorphic binary is not rewriting itself, it incurs no additional performance overhead, so the overhead is strictly the product of the time for a complete morph (e.g., one millisecond) and the frequency of morphs, as determined by the injected morph triggers. For our experiments of `dc` that

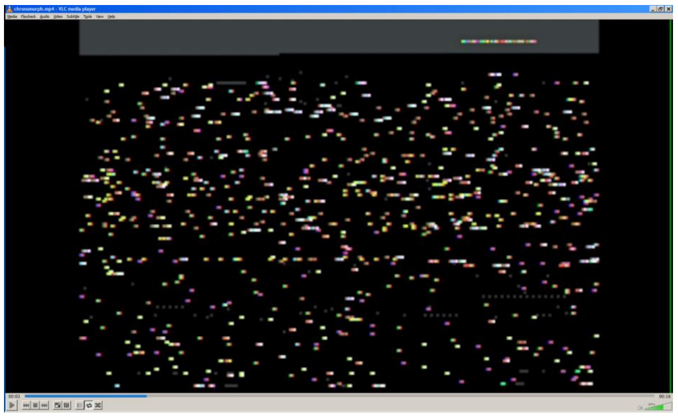
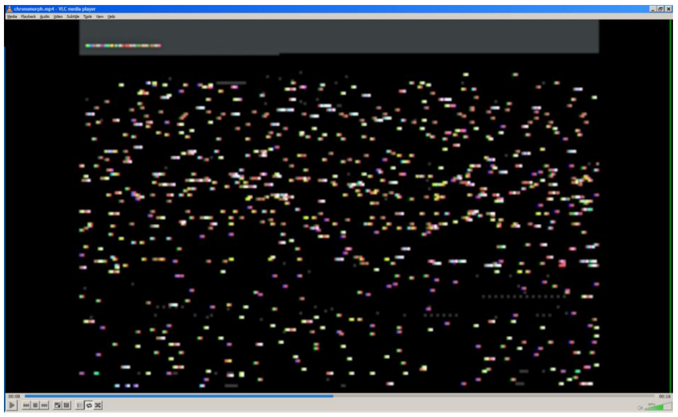



Figure 4. Example memory visualizations illustrating how the executable memory space of the binary changes at runtime.

performed a short regression test, the chronomorphic version of `dc` incurred an additional 2% overhead, but overhead will depend on morph trigger placement for other binaries. Note, however, that a more compute-intensive application might suffer a mild degradation due to cache-misses and branch prediction failures that might not occur in the non-morphing version.

Figure 4 shows two bitmaps illustrating how the binary instructions change in memory, as the program runs. Each pixel of each image represents a single byte of the program's executable code segment in memory. At the top of both images, the gray area is the `nop`-filled block relocation space, with colored segments representing the blocks moved there. Note that the colored segments are in different locations in the two images. Below the gray area, the original binary bits that are never changed remain black, while instructions that are rewritten are shown in different colors, where the RGB is computed from the byte values and `nop` instructions are gray. Again, comparison of the images will show that many of the colored areas are different between the images.

We assessed this example's morph table and estimate that it is capable of randomly assuming any one of approximately 10^{500} variants at any given time during execution. The chronomorphic version of the statically-linked target binary ($>$ 500KB) can assume any one of approximately 10^{8000} variants, using about 8ms to perform all of its rewrites. However, those variant counts do not really accurately characterize the probability that a ROP or BROP attack will succeed.

To do that, we must consider how many gadgets the attacker would need to locate, and how they are morphing. The dynamically linked target contains 250 indirect control flow instructions, and two thirds of those potentially risky elements are moved by the block-relocation phase. With the ROPgadget compiler we used for this evaluation, the original application yielded an exploit needing eight gadgets, of which six were subjected to morphing:

- `inc eax ; ret` – relocated.
- `int 0x80` – relocated.
- `pop edx ; ret` – relocated.
- `pop edx ; pop ecx ; pop ebx ; ret` – reordered (6 permutations).
- `pop ebx ; ret` – relocated.
- `xor eax,eax ; ret` – intact.
- `pop eax ; ret` – relocated.
- `move [edx],eax ; ret` – intact.

Five of the gadgets are relocated dynamically within the block relocation space of size $|b|$, and a sixth gadget is rewritten to one of six permutations. As a result, to accurately locate all eight of those gadgets in the chronomorphed binary, a potential ROP attacker would have to pick correctly from approximately $6{*}|b|^5$ alternatives. For our $|b| = 4$KB example, the probability of a correct guess is approximately $1/10^{18}$, which is extremely unlikely. Needless to say, the ROPgadget exploit was unable to compromise the chronomorphic binary, in thousands of tests. Furthermore, a BROP attack will have no ability to accumulate information about gadget locations, because they change every time a new input is received.

V. Conclusion and Future Work

We have implemented an initial version of an automatic Chronomorph tool and demonstrated that the resulting chronomorphic binaries are resistant to ROP and BROP attacks and retain their initial functionality. Our automatically-generated chronomorphic binary incurred no runtime overhead during normal operation, and only incurred one millisecond overhead to perform over 1000 sequential rewrites to executable memory during a morph operation. However, many research challenges remain for safety and scalability, including:

• **Handling threading —** The morphing behavior must not affect code blocks that are in the middle of execution. For multi-threaded applications, this will require a mechanism to lock the threads out of morphing sections or, more simply, to synchronize the threads in preparation for a morph. If source code is available, adding these sorts of mechanisms is relatively straightforward. To work on pure binaries, more powerful data flow analysis and code injection methods will be required.

• **Protecting the** `mprotect` **—** The system call that allows the chronomorphing code to rewrite executable code is, of course, a dangerous call; if an attacker could locate it and exploit it, he could rewrite the code to do whatever he wants. Therefore, we would ideally like the rewriting/SMC code itself to relocate or transform at runtime; however, the code cannot rewrite itself. We can work around this limitation with a fairly simple trick: we can use two copies of the critical code to alternately rewrite or relocate each other throughout runtime.
• **Protecting the morph table —** While chronomorphic binaries do not rely on obscurity for security, an attacker's chances of success would be higher if he has access to the morph table describing how the binary can change itself. Fairly straightforward encryption techniques should allow us hide and denature the morph table.

These challenges represent areas of future research and development for chronomorphic programs. Our prototype tool and preliminary analyses demonstrate that chronomorphic binaries reduce the predictability of code reuse attacks for single-threaded programs, and we believe that these avenues of future work will improve the safety and robustness of chronomorphic binaries in complex multi-threaded applications.

Acknowledgments

This work was supported by The Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contract FA8650-10-C-7087. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007, pp. 552–561.
[2] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 30–40.
[3] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in USENIX Security Symposium, 2011, pp. 25–41.
[4] J. Salwan and A. Wirth, "Ropgadget," URL http://shell-storm.org/project/ROPgadget, 2011.
[5] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 571–585.
[6] S. E. Friedman, D. J. Musliner, and J. M. Rye, "Improving automated cybersecurity by generalizing faults and quantifying patch performance," International Journal on Advances in Security, vol. 7, no. 3–4, in press.
[7] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 40–51.
[8] H. Shacham et al., "On the effectiveness of address-space randomization," in Proceedings of the 11th ACM conference on Computer and communications security. ACM, 2004, pp. 298–307.
[9] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in Proceedings of the 2010 workshop on New security paradigms. ACM, 2010, pp. 7–16.
[10] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 601–615.
[11] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012, pp. 157–168.
[12] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in Network and System Security. Springer, 2013, pp. 293–306.
[13] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in Proceedings of the 35th IEEE Symposium on Security and Privacy, 2014, pp. 227–242.
[14] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in USENIX Security Symposium, 2012, pp. 475–490.
[15] M. Backes and S. Nürnberger, "Oxymoron: making fine-grained memory randomization practical by allowing code sharing," in Proceedings of the 23rd USENIX conference on Security Symposium. USENIX Association, 2014, pp. 433–447.
[16] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in Information and Communications Security. Springer, 2004, pp. 187–199.